

**CoAP-Based Enablers for Designing
Efficient and Reliable Distributed IoT Applications**

**CoAP-gebaseerde bouwblokken voor de realisatie
van efficiënte en betrouwbare gedistribueerde IoT-toepassingen**

Girum Ketema Teklemariam

Promotoren: prof. dr. ir. J. Hoebeke, prof. dr. ir. I. Moerman
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen



**UNIVERSITEIT
GENT**

Vakgroep Informatietechnologie
Voorzitter: prof. dr. ir. B. Dhoedt
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2017 - 2018

ISBN 978-94-6355-085-7
NUR 986
Wettelijk depot: D/2018/10.500/3



Promotors: Prof. Jeroen Hoebeke
Prof. Ingrid Moerman

Jury Members: Prof. Luc Taerwe (Chairman)
Prof. Jeroen Hoebeke (promoter)
Prof. Ingrid Moerman (promoter)
Prof. Kris Steenhaut (Vrije Universiteit Brussel)
Prof. Johann Marquez-Barja (Universiteit Antwerpen)
Prof. Sofie Van Hoecke (EA06)
Prof. Bruno Volckaert (EA05)
Prof. David Plets (EA05)

Ghent University
Faculty of Engineering and Architecture

Department of Information Technology
iGent Tower, Technologiepark 15
B-9052 Gent, België

Tel: +32 9 331 49 00
Fax: +32 9 331 48 99
Web: <http://www.intec.ugent.be>



Dissertation to obtain the Degree of
Doctor of Computer Science Engineering
Academic year 2017 - 2018

Acknowledgements

First and for most የረዳኝን አግዚአብሔርን አመሰግናለሁ። Next, I would like to thank Piet Demeester and Ingrid Moerman for accepting my request to join the then IBCN research group so that this long journey of my PhD life starts. The help of Ingrid didn't stop there. She has been one of my promoters and was constantly inspiring me to continue with my PhD work, despite her busy schedule. I am highly indebted to my promoter Jeroen Hoebeke who has always been with me throughout my PhD study. Irrespective to my sporadic availability, due to my other engagements, he has been supporting me in every regard so that I don't get distracted and lose focus. I appreciate the way he gives me feedbacks and his ways of discussions that were important for me to push forward even in very difficult times. Jeroen: I have learned a lot from you. I would really like to say thank you very much from the bottom of my heart. My special thanks also go to the jury members Kris Steenhaut, Johann Marquez-Barja, Sofie Van Hoecke, Bruno Volckaert and David Plets for their invaluable comments and feedbacks that were used as input to improve this dissertation.

I would also like to extend my gratitude to three special people who have played an important role in my professional life. I met Rudy Gevaert in 2006 in a match-making mission organized by VLIR-UOS at the beginning of the collaborative project that sponsored my PhD study. The objective of the mission was to meet Flemish ICT professionals who will be working on the project. But our meeting resulted in much more than that. In addition to spending a successful 10 years of professional collaboration, we ended up being very good friends. He was the one who introduced me to IBCN and he was the one who wrote the Dutch summary of this dissertation. He and his wife, Natra, were always around to help me get used to Ghent and the Belgian culture (along with their two cute dogs – Zino and Pax). They have also provided accommodation for this last stay in Ghent. Rudy and Natra: I can't thank you enough. Luc Duchateau and Kora Tushune have also been of a great help during my PhD study. The support they extended from facilitating

funds for my study to their continuous encouragement was very valuable to me. This PhD wouldn't be possible without their kind help.

It is hard to pass without expressing my gratitude to my very good friend and great person Mr Kassahun Eba and the ICOS of Ghent University (Annick, Elien, Helke, Madina and Mira) for their kind assistance and flexibility in arranging my travels.

Colleagues play a significant role on ones accomplishment. I am delighted to be working with the kind and ready-to-help people at IBCN (IDLab). Floris Van Den Abeele and Isam Ishaq have been a constant help throughout my PhD study. They have contributed a lot on most of my work. I really thank you, Floris and Isam, for every contribution you made both professionally and personally. I am also grateful to Peter Ruckebusch and Bart Jooris for their contribution while I was working on two papers. I started my PhD when IBCN was at Zuiderpoort. I was staying at a student dormitory which is exactly 11 minutes walk away from my office. But I used to take two buses and reach after 40 minutes since I didn't know the routes very well. It was Wei who showed me the route and saved me lots of hustle. I would like to thank her for being so thoughtful and for all the help she has been giving me ever since. I would also like to thank Jen Rossey for his kind remarks, wonderful sense of humour and for his help in translating one part of the summary of the PhD to Dutch. I am thankful to Jetmir, Enri, Abdulkadir and Matteo for all the pet talks we had at the kitchen over coffee. Enri and Jetmir, I hope our ideas will take off and we will have something interesting in a short while. I am also thankful to Vasileios and Dries for all those days we travelled together to Sterre campus student café to escape having sandwiches at lunch (which is unusual in my culture). I can't forget my fellow countryman, Michael. It is always good to use your mother tongue and discuss about home every now and then. ስለሁሉም ነገር እጅግ ባለሁለት ስለመስጠት ምስጋና ማቅረብ፡፡ (Thank you very much for everything). I would also like to thank Xianjun, Adnan and Aslam for keeping company on our short trips to the sandwich bar and for all the ideas we exchanged along the way. I am especially thankful to Adnan for his help in my quest to figure out what should follow PhD. I don't also forget the late hour walks I used to have with Tarik and Merima after working late in office.

For someone who travels abroad, finding someone who you know back home helps a lot in getting used to life in the new culture. In this regard, I would like to thank Dawit and Ribka and their lovely kids (Daan and Eldana) for opening their home to me so that I don't feel a stranger. Natra (Rudy's Wife) and Biruk have also been a very good source of vital information. I am also grateful to Rudy's parents, Robert and Barbara. Dank u wel! We are not friends any more. We are family. ያደረጋችሁልኝን ፈጽሞ አልረሳም፡፡ እግዚአብሔር ይስጥልኝ፡፡ (I will never forget what you did for me). I would also like to extend my sincere appreciation to this lovely

family in Leuven who are originally from my hometown, Jimma, for what they did for me. Ermias and Fitsum (and their lovely kids Abigail, Yadon and Yonathan), I am so grateful for your excellent hospitality and introducing me to other friends in Leuven. Paulos and Nebiyat (and their kids), Rosa, *Woizero* Mulu, Jalle, Seble, Gelila, ... Thank you very much for all those wonderful days I spent and all the birthdays I celebrated with you.

I am also grateful to Micheline and her husband, René Morel, for letting me stay at their place whenever I come to Ghent. Due to their caring nature, I was feeling at home whenever I used to stay at their place. Thank you Zeleke for introducing me to these wonderful people. I would also like to thank my friends, Tsegaye, Sileshi, Eba, Fikremariam, and Lizet for all the cooking, the friendly chats and debates we had during our stay at Micheline's.

Last but not least, I would also like to extend my heart felt gratitude to my parents, *Ato* Ketema Teklemariam and *Woizero* Me'aza La'eke. Your life principles and prayers are the major driving forces that leads me to success. I can't repay what you have done for me. I am forever indebted. I can't finish listing all my family and friends back home who bare with me till today in order for me to be successful not only professionally but also in life. Ababu, Abrish, Ayash, Aye, Abush, Belaynesh, Bereket (and his family), Daniel (and his family), Dereje, Demis, Efrata, Emaye, Emush, Etete, Frew, Genet, Lily K., Lily D. (and her family), Habtu, Hadas, Hanna, Haimanot, Mahilet, Mame, Meron, Mimi, Muluken K, Muluken Y., Nina (and her family), Simret, Tadelech, Tiegist (and her family), Yodit, Yonathan, (and many more). I am very lucky to be a member of this wonderful family. Thank you all. My great appreciation also goes to my colleagues at Jimma University, Amanuel, Berhanu, Basiliyos, Hunde, Shimels, Wondu and Zegeye who had been working on my behalf during my absence. I would also like to thank Hiwot for helping me with rearranging data when I was working on conditional observation.

I saved the best for last. This PhD study would not be possible without the help and support of my beautiful wife and partner in life, Yemsrach Alemayehu. She was my strength during all the darkest hours. She has been taking care of our wonderful children, Natan and Mathias, during my absence. I would like to say "You have an exceptional place in my heart. I am sure the future is bright with you. Thank you very much my dear." Of course, the help of Shitaye and Amerke in taking care of the children is also exceptional.

My happiness would have been complete if my elder brother, Addisu Ketema (Nov 27, 1971 – Nov 27, 2016), would have been with me today. He has sacrificed his youth so that his younger siblings become successful. Even in the last moments, he was too careful not to disturb my study and prevented my family from informing me of his illness. You are always my inspiration. I am thankful for everything you did for me my beloved brother. እግዚአብሔር ነፍሱን በገነት ያኑራት፡፡

Ghent, December 2017
Girum Ketema Teklemariam

Table of Contents

Acknowledgements	i
List of Acronyms	xv
Samenvatting – Summary in Dutch –	xxi
English Summary.....	xxvii
1 Introduction	1
1.1 The Internet of Things (IoT).....	1
1.2 IoT Systems	3
1.2.1 Components of Generic IoT Systems	4
1.2.2 Connectivity and communication	5
1.2.3 Data Processing	7
1.3 Application development paradigms	8
1.3.1 Cloud-Centric Architecture.....	8
1.3.2 Gateway Centric Architecture	8
1.3.3 Distributed Architecture	8
1.4 Challenges in Building IoT Applications	9
1.4.1 Collection of Redundant Data.....	9
1.4.2 All Intelligence on Non-Constrained Device or in the Cloud	10
1.4.3 Static Configurations	11
1.4.4 Unexpected Crashes of Constrained Devices	11
1.5 Research Contributions.....	12
1.6 Fit within the Broader IoT Landscape.....	17
1.7 Outline	20
1.8 List of publications	22
1.8.1 A1 publications (listed in the Science Citation Index).....	22

1.8.2 Publications in other International Journals.....	23
1.8.3 Publications in International Conferences (listed in the Science Citation Index)	23
1.8.4 Publications in international conferences	23
1.8.5 Patent Applications	24
References.....	25
2 Facilitating the creation of IoT applications through conditional observations in CoAP	27
2.1 Introduction	28
2.2 The Constrained Application Protocol and Observe.....	30
2.3 Related work	35
2.4 Conditional Observe.....	36
2.4.1 The Condition Option Format.....	38
2.4.2 Condition Types	38
2.5 Implementation.....	43
2.6 Evaluation	46
2.6.1 Scenario 1: Basic Evaluation	46
2.6.2 Scenario 2: Non-constrained Client - Gateway – Multiple servers.....	48
2.6.3 Mathematical Evaluation	49
2.7 Use Cases	55
2.7.1 Heating and Cooling Systems.....	56
2.7.2 Smart Environment Monitoring.....	57
2.7.3 Sleepy Nodes	59
2.8 Conclusion	60
References.....	61
3 Bindings and RESTlets: A Novel Set of CoAP-Based Application Enablers to Build IoT Applications.....	63
3.1 Introduction	64
3.2 Constrained Application Protocol (CoAP)	67
3.3 Related Work	71
3.3.1 Sensor-Actuator Interaction	71
3.3.2 In-Network Processing	71
3.4 Flexible Direct Binding.....	73
3.5 RESTlets	77
3.6 Implementation and Evaluation	81
3.6.1 Implementation	81
3.6.2 Experiment Setup.....	83

3.6.3	Functional Evaluation.....	84
3.6.4	Performance Evaluation.....	86
3.7	Conclusions and the Way Forward	100
References.....		102
4	Dynamic Deployment of RESTlets on Constrained Devices	105
4.1	Introduction	106
4.2	Related Work	108
4.3	RESTlets and Bindings.....	109
4.3.1	RESTlets.....	109
4.3.2	Bindings.....	110
4.3.3	Example 1 – Sample IoT Application.....	112
4.3.4	Example 2 – RESTlets and Conditional Observe	113
4.4	Architecture and Dynamic Deployment of RESTlets.....	115
4.5	Implementation and Evaluation	120
4.5.1	Implementation of Dynamic RESTlets on Constrained Devices	120
4.5.2	Implementation of Conditional Observe using Dynamic RESTlets ...	125
4.5.3	Functional Evaluation.....	126
4.5.4	Performance Evaluation.....	127
4.6	Conclusion and Future work	138
References.....		140
5	Transparent Recovery of Dynamic States on Constrained Nodes through Deep Packet Inspection.....	143
5.1	Introduction	144
5.2	Related Work	146
5.3	Dynamic States and State Recovery.....	147
5.3.1	Dynamic States	147
5.3.2	Recovery of Dynamic States.....	151
5.4	Dynamic State Recovery	152
5.4.1	Transparent Dynamic State Recovery for Unencrypted Communication 153	
5.4.2	Transparent Dynamic State Recovery for Encrypted Communication	159
5.4.3	Other Dynamic State Recovery Mechanisms	160
5.5	Implementation.....	162
5.6	Evaluation	164
5.6.1	Functional Evaluation.....	164
5.6.2	Performance Evaluation.....	169
5.7	Conclusion and Way Forward.....	172

Reference	174
6 Conclusion	177
6.1 Summary and Conclusion	178
6.2 Future work.....	181

List of Figures

Figure 1-1: IoT Application Domain	2
Figure 1-2: Generic IoT System	4
Figure 1-3: IETF Protocol Stack.....	5
Figure 1-4 CoAP Client-Server Interactions.....	7
Figure 1-5: Collection of Redundant Data.....	9
Figure 1-6: Sensor (Light Switch) and Actuator (Light Bulb) Interaction through an Intermediary	10
Figure 1-7: Effect of Rebooting Nodes on IoT Applications.....	12
Figure 1-8: Contribution: Conditional Observe	13
Figure 1-9: Contribution: Bindings (LS=Light Switch and LB = Light Bulb)	14
Figure 1-10: Contribution: RESTlets	15
Figure 1-11: Contribution: Dynamic Loading	16
Figure 1-12: Contribution: Crash Recovery.....	17
Figure 2-1: CoAP Message Format consisting of a 4-bytes base binary header followed by optional extensions	31
Figure 2-2: CoAP option format	32
Figure 2-3: CoAP Client/Server communication	32
Figure 2-4: Normal Observation	33
Figure 2-5: Conditional Observation	37
Figure 2-6: Format of the option value of the Condition Option	38
Figure 2-7: Temperature (°C) Data over 120 Seconds	39
Figure 2-8: Notifications Generated While Using Different Condition Types.....	39
Figure 2-9: Architecture of Erbium	43
Figure 2-10: Conditional Observation Module	44
Figure 2-11: Number of Packets transmitted Vs. Hop count	47
Figure 2-12: Power Consumption Vs. Hop Count (Non Confirmable Transmission).....	47

Figure 2-13: Power Consumption of nodes (Confirmable Transmission) ..	48
Figure 2-14: Experimental setup consisting of non-constrained client, and 2 constrained servers	48
Figure 2-15: Power Consumption Vs. Probability of Packet Transmission	52
Figure 2-16: Distribution of the power consumption over all different energy consumers.....	53
Figure 2-17: Reduction in energy consumption of using conditional observations with $p=0.75$ versus normal observe for a varying number of resources on the server.....	54
Figure 2-18: Experiment setup with 2 Click++ clients, a Contiki border router, a Contiki Server and 2 Intermediate nodes.	57
Figure 2-19: Air Quality Controlling Setup	58
Figure 2-20: Communication with sleepy nodes using conditional observation	59
Figure 3-1: (a) CoAP GET Operation; (b) CoAP PUT Operation.	68
Figure 3-2: CoAP Header	69
Figure 3-3: CoAP Observe Operation.....	70
Figure 3-4: Sensor-Actuator Interaction. (a) Indirect; (b) Direct Binding. .	74
Figure 3-5: Flow Chart Showing Binding Relationship Establishment.	75
Figure 3-6: Flow Chart Showing Notification of Events.	76
Figure 3-7: RESTlet Block Diagram.	77
Figure 3-8: Sample Code Executed on Non-constrained Devices	79
Figure 3-9: RESTlet block diagram for the smart home scenario.....	80
Figure 3-10: CoAP Messages used to create the required Binding Relationship	80
Figure 3-11: Flowchart Showing Interaction with RESTlet Instances using CoAP Messages	83
Figure 3-12: Creation of Binding Using CoAP++ GUI from Non- constrained Device	84
Figure 3-13: Direct Interaction of Sensor and Actuator Nodes in Cooja....	85
Figure 3-14: Creation of RESTlet Instances in Copper	86
Figure 3-15: Topologies: (a) Sensor and actuator in different branch of the tree; (b) Actuator between Sensor and Gateway—directly connected; (c) Actuator between Sensor and Gateway after 1 hop; (d) Sensor between Actuator and Gateway after 1 hop. ...	89
Figure 3-16: Communication Delay (ms) vs. Topology	90
Figure 3-17: Sensor-Actuator Interactions. (a) Binding (b) Gateway/Cloud- Based Solution.	91
Figure 3-18: Network Topology	92
Figure 3-19: Packet Processing and Forwarding time at RESTlet Node for Various Number of Data Generating Nodes	95

Figure 3-20: Impact of Number of Data Generating Nodes on End-to-End Latency. (a) Confirmable Communication; (b) NON-Confirmable Communication.	96
Figure 3-21: Network Topology including a Node Generating Side Traffic.....	97
Figure 3-22: Impact of Side Traffic on Latency. (a) CONfirmable; (b) NON-Confirmable transaction	98
Figure 3-23: Impact of Packet Arrival Time Gap on Latency	99
Figure 3-24: Network Topology for Noisy Networks.....	100
Figure 3-25: Impact of TX/RX Reception Ratio on Latency. (a) CONfirmable Communication; (b) NON-Confirmable Communication.	100
Figure 4-1 RESTlet.....	110
Figure 4-2: Binding Relationship Establishment	111
Figure 4-3: RESTlet-based IoT Application	112
Figure 4-4: Statements to Create RESTlet-based IoT Application	113
Figure 4-5: Conditional Observe Option Format	114
Figure 4-6: Conditional Observation Processing Logic Flow-chart	114
Figure 4-7: Generic IoT System	116
Figure 4-8: Generic RESTlet Architecture	116
Figure 4-9: Implementation Options of Generic RESTlet Architecture. a) Static. b) Template Based. c) Dynamic. d) Hybrid	119
Figure 4-10: Implementation of Dynamic RESTlets on Constrained Devices.....	121
Figure 4-11: Instructions from MSP430 Makefile.....	124
Figure 4-12: Block Diagram of Implementation of Conditional Observe Using Dynamic RESTlets	125
Figure 4-13: Compiling Dynamic RESTlet (CND)	126
Figure 4-14: Cooja Simulation Showing Transfer of the last Blocks	127
Figure 4-15: Contribution of RESTlet Architectural Components to the Overall Memory Footprint	129
Figure 4-16: Dynamic RESTlet Deployment Time	131
Figure 4-17: Energy Usage for Dynamic Deployment. a) No RDC Protocol. b) ContikiMAC RDC	134
Figure 4-18: Time required to regain power consumed during dynamic deployment.....	137
Figure 4-19: Time required to regain power consumed during dynamic deployment (Max-age = 60s)	138
Figure 5-1: CoAP Interaction - PUT Request.....	148
Figure 5-2: CoAP Interaction - Observation Request	149
Figure 5-3: CoAP Interaction - Binding Request.....	150
Figure 5-4: CoAP Interaction – Runtime Deployment	150

Figure 5-5: Dynamic State Restoration Cycle	151
Figure 5-6: Placement of State Directory at the LLN Gateway	154
Figure 5-7: Dynamic State Information Collection	156
Figure 5-8: Observe Request Information Collection	157
Figure 5-9: Dynamic State Recovery with State Directory on the Node ..	161
Figure 5-10: Implementation of Transparent Dynamic State Recovery ...	162
Figure 5-11: Registration of a node at the gateway	164
Figure 5-12: Dynamic State Collection (using PUT method).....	164
Figure 5-13: Intercepted PUT Request from Client.....	165
Figure 5-14: CoAP Observe Request Sent from Copper Client.....	166
Figure 5-15: Interception of a New Observe Request at the SD	167
Figure 5-16: List of Entries in the SD Before Node [aaaa::c30c:0:0:2]....	167
Figure 5-17: SD Containing 3 Records.....	168
Figure 5-18: Output of the node after receiving the 3 packets from the SD	168
Figure 5-19: CoAP Message on Copper Showing Resumption of the Observe Operation	169
Figure 5-20: Delay Introduced by SD-Node Association Process	169
Figure 5-21: Impact of Hop Count on Recovery Delay	171
Figure 5-22: Impact of Number of Dynamic States that needs to be recovered on recovery time	172

List of Tables

Table 2-1: Memory requirements of Normal and conditional observe	45
Table 2-2: Parameter Values.....	51
Table 3-1: Memory Foot Print (Byte).	87
Table 4-1: RESTlet Structure	122
Table 4-2: Information Required to Transfer the Dynamic Module	126
Table 4-3: Firmware Size (Byte)	128
Table 4-4: Sizes of RESTlet Architectural Components	129
Table 4-5: Number of Packets Required to Transfer RESTlets	130
Table 4-6: Power Specification of Zolertia Z1 Mote	133
Table 4-7: Energy consumed for Dynamic Deployment	136

List of Acronyms

#

3GPP	3 rd Generation Partnership Project
6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
6TiSCH	IPv6 over the TSCH mode of IEEE 802.15.4e

A

API	Application Programming Interface
-----	-----------------------------------

B

BLE	Bluetooth Low Energy
-----	----------------------

C

CoAP	Constrained Application Protocol
CoRE	Constrained RESTful Environments
CPU	Central Processing Unit

D

DODAG	Destination Oriented Directed Acyclic Graph
DTLS	Datagram Transport Layer Security

E

EC-GSM-IoT	Extended coverage GSM Internet of Things
ETSI	European Telecommunication Standards Institute

H

HTTP	Hyper Text Transfer Protocol
------	------------------------------

I

IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
IoT	Internet of Things
IP	Internet Protocol
IPSO	Internet Protocol for Smart Objects
IPv6	Internet Protocol Version 6
ITU	International Telecommunications Union

L

LLN	Low-power and Lossy Network
LoRa	Long Range Radio
LoRaWAN	LoRa Wide Area Network

LPM	Low Power Mode
LPWAN	Low Power Wide Area Network
LTE	Long-Term Evolution
LTE-M	Long-Term Evolution Category M1

M

M2M	Machine to Machine
MAC	Medium Access Control
MEMS	Micro-Electro-Mechanical-Systems
MQTT	Message Queuing Telemetry Transport
MTU	Maximum Transmission Unit

N

NB-IoT	NarrowBand Internet of Things
--------	-------------------------------

O

OCF	Open Connectivity Forum
OGC	Open Geospatial Consortium
OMA LWM2M	Open Mobile Alliance Light Weight Machine-to-Machine
OSGi	Open Service Gateway Initiative
OSI	Open Systems Interconnection

R

RAM	Random Access Memory
RDC	Radio Duty Cycling

REST	Representational State Transfer
ROLL	Routing over Low-power and Lossy Network
ROM	Read Only Memory
RPL	Routing Protocol for Low-Power and Lossy Networks
RX	Receive

S

SD	State Directory
SMS	Short Message Service
SOAP	Simple Object Access Protocol
SOS	Sensor Observation Service
SSL	Secure Sockets Layer

T

TCP	Transmission Control Protocol
TLS	Transport Layer Security
TSCH	Time Slotted Channel Hopping
TX	Transmit

U

UBN	Ultra-Narrow Band
UDP	User Datagram Protocol
uIP	Micro Internet Protocol
URI	Universal Resource Identifier

W

WSN Wireless Sensor Network

X

XML eXtensible Markup Language

Samenvatting

– Summary in Dutch –

In het laatste decennium van de twintigste eeuw is het Internet exponentieel gegroeid door de interconnectie van miljarden toestellen. Op dit moment kent het Internet terug een periode van extreme groei. Dit fenomeen is grotendeels te wijten aan de evolutie in draadloze netwerken en elektromechanische technologieën. De technologische vooruitgang in beide domeinen opende nieuwe mogelijkheden om mobiele telefoons en slimme objecten essentiële componenten te laten worden van ons huidige Internet. De toekomst zal een periode worden van verdere uitbreiding. Volgens Cisco zullen er 50 miljard geconnecteerde toestellen zijn tegen 2020. De meeste van deze toestellen zullen slimme objecten zoals sensoren en actuatoren zijn. Sensoren lezen natuurlijke gegevens uit, zoals de temperatuur en vochtigheid van de omgeving, en verzenden de gegevens naar een meer geschikt toestel voor verdere verwerking. Actuatoren daarentegen wijzigen hun omgeving gebaseerd op de ontvangen input van andere toestellen. Sensoren en actuatoren kunnen geplaatst worden op eender wat, gaande van grote structuren tot zelfs op lichaamsdelen. Deze evolutie heeft bijgedragen tot het tot stand komen van een nieuw domein, het Internet der Dingen (Engels: Internet of Things, IoT). Een voorname eigenschap van het IoT is de interconnectie van zeer heterogene toestellen uit verschillende domeinen door middel van diverse technologieën.

De slimme objecten die deel uitmaken van het Internet hebben inherente beperkingen. Ze hebben beperkingen inzake geheugen en rekenkracht. Hun communicatiemogelijkheden zijn ook sterk beperkt. Bovendien werken ze op batterijen waardoor energiebesparende methodes gebruikt moeten worden om de batterijduur te verlengen. Door deze beperkingen worden de netwerken die gevormd worden door de interconnectie van deze toestellen ook wel laag-energie en verliesgevende netwerken genoemd (Engels: low power and lossy networks, LLNs). Deze eigenschappen maken het onmogelijk om de bestaande Internet communicatieprotocollen te gebruiken. Dat bracht verschillende spelers ertoe om nieuwe oplossingen te vinden om deze toestellen met het Internet te verbinden. Sommige bedrijven gebruiken een tussenliggend toestel, ook wel gateway genoemd, om het LLN te interconnecteren met het Internet. Propriëtaire

communicatieprotocollen worden dan gebruikt om de slimme objecten te interconnecteren met deze gateway, waarbij deze laatste de vertaling maak naar standaard Internetprotocollen. Dergelijke aanpak resulteerde in verticale silo's die niet interoperabel zijn en niet leiden tot de realisatie van één groot netwerk waarin iedereen met elkaar kan praten.

Sommige standaardisatiegroepen, zoals het IETF en het IEEE, namen daarom de leiding in het definiëren van open standaarden die gebruikt kunnen worden als leidraad bij het ontwikkelen van de IoT infrastructuur. Bijvoorbeeld, IEEE 802.15.4 voorziet de fysieke en MAC-laag specificaties die draadloze communicatie met een laag energieverbruik toelaten over een beperkte afstand. Het werk van het IETF spitst zich toe op het gebruiken van IP-gebaseerde eind tot eind communicatie om toestellen toegankelijk te maken vanaf het Internet. Dat leidde tot de definitie van de 6LoWPAN adaptatielaag die toeliet IPv6 pakketten te routeren binnenin LLNs terwijl voldaan was aan de beperkingen opgelegd door de lagere protocollagen. Routing wordt aangepakt door de introductie van een nieuw routingsprotocol voor lage-energie en verliesgevende netwerken (Engels: Routing Protocol for Low-Power and Lossy Networks (RPL)). De applicatielaag is geen uitzondering. Een nieuw applicatieprotocol voor constrained devices, het Constrained Application Protocol (CoAP), werd geïntroduceerd door het IETF als een lichtgewicht versie van HTTP. De combinatie van deze protocollen laat op een naadloze manier eind tot eind communicatie toe tussen toestellen op het Internet (zoals een smartphone) en objecten met heel beperkte mogelijkheden (bijvoorbeeld een pacemaker), zonder dat er nog een specifieke gateway nodig is.

Dit doctoraat concentreert zich op het gebruiken van gestandaardiseerde oplossingen om bijkomende mogelijkheden en meer flexibiliteit te voorzien voor het IoT. Meer specifiek, het volledige doctoraat concentreert zich rond CoAP en zijn extensies. We stellen verschillende uitbreidingen voor aan het CoAP protocol en testen de efficiëntie van deze nieuwe oplossingen. De eerste bijdrage van dit doctoraat is de uitbreiding van het CoAP observe-mechanisme om de efficiëntie te verbeteren. Het bestaande CoAP observe-mechanisme laat toestellen of applicaties, ook wel clients genoemd, toe om hun interesse in statusveranderingen van CoAP resources die aangeboden worden door een ander toestel, ook wel server genoemd, kenbaar te maken. Ze sturen dan een GET-verzoek dat een observe-optie bevat naar de server. Eenmaal geregistreerd, verwittigt de server de client van elke verandering in de toestand van de resource zoals bijvoorbeeld de waarde gemeten door een temperatuursensor. Vaak is dit inefficiënt omdat niet alle statusveranderingen relevant zijn voor de applicatie. Dit leidt tot een verspilling van kostbare bandbreedte en energie. In dit doctoraat stellen we een oplossing voor op basis van conditionele observaties. Dit laat clients toe om

notificatiecriteria toe te voegen bij het registratieverzoek. Op die manier zal de server geen notificaties meer sturen tenzij ze voldoende aan de notificatiecriteria.

De tweede bijdrage van dit doctoraat bekijkt nieuwe mechanismen die het bouwen van CoAP-gebaseerde applicaties makkelijker maken. De twee voorgestelde mechanismen zijn Bindings en RESTlets. Bindings zijn CoAP observe-relaties die tot stand gebracht worden door een derde partij. In de huidige protocolspecificatie creëert een GET-verzoek met de observatie-optie een observe-relatie tussen de zender en de ontvanger van het verzoek. Omwille van dit feit worden interacties tussen een sensor en een actuator typisch tot stand gebracht via een tussenliggend toestel dat altijd online moet zijn. Het tussenliggende toestel brengt een observe-relatie tot stand met de sensor, verwerkt de resulterende notificaties en stuurt indien nodig een signaal naar de actuator. Deze aanpak heeft verschillende nadelen zoals bijkomende vertragingen in LLNs, teveel verkeer en pakketverlies aan de grens van het LLN en het falen van het systeem wanneer het tussenliggende toestel wegvalt. Bindings laten de creatie toe van directe interacties tussen sensoren en actuatoren door toe te staan dat een derde partij de observe-relatie tot stand kan brengen. Om dat te doen introduceren we vier nieuwe CoAP opties die de server toelaten het onderscheid te maken tussen normale observe-verzoeken en binding-verzoeken. Bij het ontvangen van het verzoek registreert de server de actuator als een observator in plaats van de initiator. Wanneer de relatie ingesteld is, is de initiator niet meer betrokken bij verdere communicatie. In tegenstelling tot het maken van een relatie tijdens het compileren, wat zeer inflexibel is, laat deze aanpak toe dat IoT applicaties vrije associaties aangaan op ieder moment zonder beperkingen.

Het tweede mechanisme dat wordt voorgesteld zijn RESTlets. RESTlets zijn IoT applicatiebouwstenen die input ontvangen, verwerken en output produceren. Ze hebben ook controleparameters die gebruikt kunnen worden om de configuratieparameters bij te regelen. De input kunnen sensormetingen zijn of de output van een andere RESTlet. De output kan dan weer gebruikt worden als input voor actuatoren, IoT componenten in de gateway of in de cloud, of zelfs voor andere RESTlets. De verwerkingslogica kan zeer simpel zijn, zoals het gemiddelde berekenen van de input, of zeer complex zoals het versturen van een sms-bericht naar een specifieke bestemming. RESTlets kunnen éénmaal gemaakt worden en verschillende keren geïntanceerd worden om verschillende verwerkingstaken uit te voeren. Een belangrijke eigenschap van RESTlets is hun plaatsing. Afhankelijk van hun complexiteit kan een RESTlet ondergebracht worden op een toestel met of zonder beperkingen. De IoT applicatie logica kan hiermee opgedeeld worden in kleinere onderdelen en gedistribueerd worden als RESTlets over verschillende IoT componenten. Bijvoorbeeld, indien we een signaal wensen te sturen naar een thermostaat gebaseerd op het gemiddelde van

vijf temperatuursensoren, kunnen we de RESTlet op één van de sensoren onderbrengen of op de thermostaat zelf waarbij deze voorzien is van vijf inputs (één voor elke sensor) en een output. Elke input brengt een observe-relatie tot stand met elke temperatuursensor zodat elke temperatuursverandering gerapporteerd wordt aan de node die de RESTlet herbergt. De actuator zal een observe-relatie hebben met de output van de RESTlet. Bij het ontvangen van een pakket zal de RESTlet de verwerking doen en de actuator een signaal geven als de output verandert. De RESTlet controleparameters kunnen gebruikt worden om sommige configuratieparameters, zoals de notificatiedrempel, te wijzigen. Om de flexibiliteit te handhaven van IoT applicaties worden Bindings gebruikt om dynamisch observe-relaties tussen componenten aan te maken.

Eén van de nadelen van de originele RESTlet benadering is dat ze statisch aangemaakt worden. Dat wil zeggen dat ze waardevol geheugen verspillen wanneer ze niet gebruikt worden. Om dit te verbeteren, hebben we het werk uitgebreid om RESTlets dynamisch te laden tijdens looptijd. We vormen elke RESTlet om tot een dynamisch laadbare module zodat deze kan worden geüpload wanneer nodig. Om functioneel te zijn dient de firmware van iedere node dynamisch laden en linken te ondersteunen. Dat is de derde bijdrage van dit doctoraat.

Tenslotte stellen we een oplossing voor die het herstellen van dynamisch gecreëerde toestandsinformatie op een transparante manier toelaat door pakketten te inspecteren. Zoals reeds aangehaald, de van nature uit beperkte toestellen zijn onbetrouwbaar. Hierdoor kan het gebeuren dat een toestel herstart zonder aanwijsbare reden. Wanneer ze herstarten en terug online komen, leidt dit tot het verlies van alle statusinformatie die gecreëerd werd door interacties met andere toestellen en opgeslagen werd in het volatiel geheugen. Toestandswijzigingen van CoAP resources, observatie-relaties, binding-relaties aangemaakt door externe toestellen en dynamisch geladen code zijn voorbeelden van dynamische statussen. IoT applicaties die gebruik maken van de herstarte nodes zullen foutieve resultaten verkrijgen of zullen niet werken. Als onderdeel van het doctoraat introduceren we een methode om deze dynamische toestandsgegevens te herstellen zonder de interventie of kennis van de partijen die deelnemen aan de communicatie. In de voorgestelde oplossing maken we een toestandsindex aan op de LLN-gateway die opgevuld wordt door ieder pakket te onderscheppen dat door het netwerk gaat en na te gaan of dit resulteert in de creatie van nieuwe toestandsinformatie. Ieder pakket dat het potentieel heeft om een dynamische toestand aan te maken op een beperkte node, resulteert in de creatie of aanpassingen van de toestandsindex. Als een beperkte node rapporteert aan de gateway dat hij aan het opstarten is, zal de gateway de toestandsindex raadplegen en een sequentie van acties starten om de dynamische toestand te hercreëren op de node.

Het doel van dit doctoraat was het gebruik van gestandaardiseerde protocollen voor IoT applicaties te onderzoeken en verschillende verbeteringen aan deze protocollen voor te stellen. Meer specifiek, we bestudeerden CoAP (en zijn observe-extensie), een protocol dat meer en meer gebruikt wordt voor de interactie met toestellen met beperkte mogelijkheden.

English Summary

In the 1990s, the Internet saw an unexpected exponential growth by interconnecting billions of devices in just a decade. Currently, it is going through yet another era of extreme growth. This phenomenon is mainly due to the evolution of wireless networking and electromechanical technologies. These two technological advancements opened up new possibilities for mobile phones and smart objects to become a vital component of today's Internet. Even the future is believed to be a period of extreme expansion. According to Cisco, the number of connected devices will reach 50 billion by 2020. Most of these devices will be smart objects such as sensors and actuators. Sensors read physical data such as temperature and humidity from their environment and transfer the data to a more capable device for further processing. Actuators, on the other hand, alter their environment based on inputs received from other devices. These sensors and actuators can be fit on anything, from huge structures to the smallest body parts. This evolution has given rise to a whole new area called the Internet of Things (IoT). A key characteristic of IoT is the interconnection of very heterogeneous devices from different domains using diverse technologies.

The smart objects that are being incorporated in the Internet have inherent constraints. They have limited memory and processing power. Their communication capabilities are also seriously limited. Moreover, they are battery operated and need power saving mechanisms to extend their battery life. Due to these constraints, the networks formed by the interconnection of such devices are called low power and lossy networks (LLNs). These characteristics make it impossible for existing Internet communication protocols to be directly used by the smart objects. As a result, several approaches have been taken by different stakeholders to realize the connection of these devices to the Internet. Some vendors have used intermediaries to interconnect the LLN with the Internet by using proprietary communication protocols to interconnect the smart objects to the intermediary and letting the intermediary perform the translation to standard-based Internet protocols. This has resulted in vertical silos which are not interoperable and move away from creating a single interconnection of devices.

Some standardization bodies, such as IETF and IEEE took the lead in defining open standards that can be used to guide the development of the IoT infrastructure. For instance, IEEE 802.15.4 provides the physical and MAC layer specifications to enable low-power wireless connectivity over limited distances. The work of IETF focuses on using IP-based end-to-end communication in order to enable devices to be accessible directly from the Internet. This has led to the definition of the 6LoWPAN adaptation layer that enables IPv6 packets to be routed within LLNs while meeting the stringent size requirements imposed by the lower protocol layers. Routing is also addressed by introducing Routing Protocol for Low-Power and Lossy Networks (RPL) as a routing protocol. The application layer is not an exception either. The Constrained Application Protocol (CoAP) was introduced by IETF as a light-weight version of HTTP. The combination of these protocols enables end-to-end communication between a device on the Internet (e.g. a smartphone) with a highly constrained smart object (e.g. pacemaker) in a seamless way, without any intermediaries.

This PhD work focuses on using standardized solutions to provide additional features and give more flexibility to the IoT. More specifically, all of the PhD work is centered on CoAP and its extension. We propose several extensions to the CoAP protocol and test the efficiency of the new solutions. The first contribution of this PhD consists of an extension of the CoAP observe mechanism in order to improve efficiency. The existing CoAP Observe mechanism allows clients to register their interest in state changes of CoAP resources hosted by a server by sending a GET request that includes the observe option. Once registered, the server (usually sensors), notifies the client of every change. However, this is often quite inefficient as not all state changes are relevant for an application, leading to a waste of precious bandwidth and energy. The conditional observation solution we propose as part of this PhD work lets clients send notification criteria along with the registration request. By doing so, the server will not send notifications unless they meet the notification criteria.

The second contribution of this PhD introduces enablers for CoAP-based IoT application development. The two enablers proposed are Bindings and RESTlets. Bindings are CoAP Observe relationships established by a third-party. In the current protocol specification, a GET request with the observe option creates an observation relationship between the sender and the receiver of the request. Due to this, interactions between a sensor and actuator are typically established via an intermediary that must be online at all times. The intermediary establishes an observe relationship with the sensor, processes the resulting notifications and sends a trigger to the actuator if required. This approach has several drawbacks such as high latency in LLNs, congestion at the border of the LLN and failure of the system in case the intermediary fails. Bindings enable the creation of direct

interactions between sensors and actuators by letting a third party establish an observation relationship between them. In order to do so, we introduce four CoAP Options that let the server distinguish between normal observe requests and binding requests. Upon reception of the request, the server registers the actuator as an observer rather than the initiator. Once this relationship is established, the initiator is not involved in further communication. As opposed to creating the relationship at compile time, which is very inflexible, this approach enables IoT applications to freely create associations at any time without limitations.

The second enabler that is introduced are RESTlets. RESTlets are IoT application building blocks that receive inputs, process them and produce outputs. They also have control parameters that can be used to tweak configuration parameters. The inputs can be sensor readings or outputs of another RESTlet, while the outputs can be fed into actuators, IoT components at the gateway or in the cloud, or even to other RESTlets as input. The processing logic can be as simple as averaging of inputs or as complex as sending an SMS to a particular destination. RESTlets can be created once and instantiated multiple times in order to perform multiple processing tasks. The most important characteristic of RESTlets is their placement. Depending on their complexity a RESTlet can be hosted on a constrained node or at non-constrained devices. In addition, IoT application logic can be broken down into smaller units and distributed as RESTlets across the different IoT components. For instance, if we need to trigger a thermostat based on the average values of 5 temperature sensors, we host the RESTlet on one of the sensors or on the thermostat having 5 inputs (one from each sensor) and an output. Each input, then establishes an observation relationship with each temperature sensor so that each temperature change is reported to the node hosting the RESTlet. The actuator will also have an observation relationship with the output of the RESTlet. Upon receiving a packet, the RESTlet does the processing and triggers the actuator if the output changes. The RESTlet's control parameter can be used to modify some configuration parameters such as a notification threshold. To maintain the flexibility of the IoT applications, bindings are used to dynamically create the observation relationships between the components.

One of the drawbacks of the original RESTlets approach is the fact that they are created statically, meaning that they are wasting precious memory when not used. To improve this, we extend the work to dynamically load RESTlets at run-time. We make each RESTlet as a dynamically loadable module so that it can be uploaded whenever required. For this to be functional, the firmware of each node must have dynamic loading and linking capability. This is the third contribution of this PhD.

Finally, we propose a solution for the transparent recovery of dynamically created state information through deep packet inspection. As repeatedly said, the constrained devices are unreliable and may go through reboot cycles for unspecified reasons. When they reboot and come back online, they have lost all state that has been created through interaction with other nodes and that has been stored in volatile memory. Examples of such dynamic state are actuation modifications, observation relationships, binding relationships created by external devices, and dynamically loaded code. IoT applications that make use of the rebooting nodes will get incorrect results or may not work at all. As part of this PhD, we introduce a mechanism for recovering the dynamic state data without the intervention or knowledge of the communicating parties. In the proposed solution, we create a state directory at the LLN gateway which is populated by intercepting each packet that traverses the network. Every packet that has the potential of creating dynamic state on a constrained node creates or modifies entries in the state directory. If a constrained node reports to the gateway that it is booting up, the gateway consults its state directory and initiates a sequence of activities to regenerate the dynamic states on the node.

To conclude, this PhD emphasizes the use of standardized communication protocols for IoT applications and suggests several improvements to the protocols. Particularly, we leverage on CoAP and its observe extension, which is witnessing extremely fast adoption for M2M communications ranging from application data communication to device management.

1 Introduction

“We need to get smarter about hardware and software innovation in order to get the most value from the emerging Internet of Things.”

– Henry Samueli (1954 -)

This chapter introduces the background of this dissertation and gives an overview of the work done. This includes an overview of the challenges addressed by the work and the contributions that have been made. Finally, it also includes a list of publications authored.

1.1 The Internet of Things (IoT)

For decades, the Internet was considered to be the interconnection of computers and other electronic devices such as scanners and printers. Several technological advances such as improved electromechanical technologies, expansion and evolution of wireless communication technologies and cloud services paved the way for a very different Internet in terms of interconnected devices. Today, smartphones, sensors and actuators have become an integral part of the Internet. Sensors read physical data such as temperature and humidity from their environment and transfer this data to the virtual world, consisting of services on

the Internet. Actuators, on the other hand, alter their environment based on inputs they receive via the Internet. These sensors and actuators can be fit on anything, from huge structures to the smallest body parts. Originally, these devices were second class citizens, only connected via proprietary gateways. Today, they are no longer merely an add-on. The term *Internet of Things (IoT)* is used to refer to this phenomenon where everything is connected to everything to share resources [1.1]. ITU-T defines IoT as an infrastructure that interconnects virtual and physical things using existing and future technologies [1.2]. ITU-T stresses that interoperability is important in IoT. So, the IoT vision is the interconnection of people, things and data across domains using a plethora of technologies to bridge the gap between the physical, digital and virtual worlds. The application domains are diversified and cross-domain communications are very common.

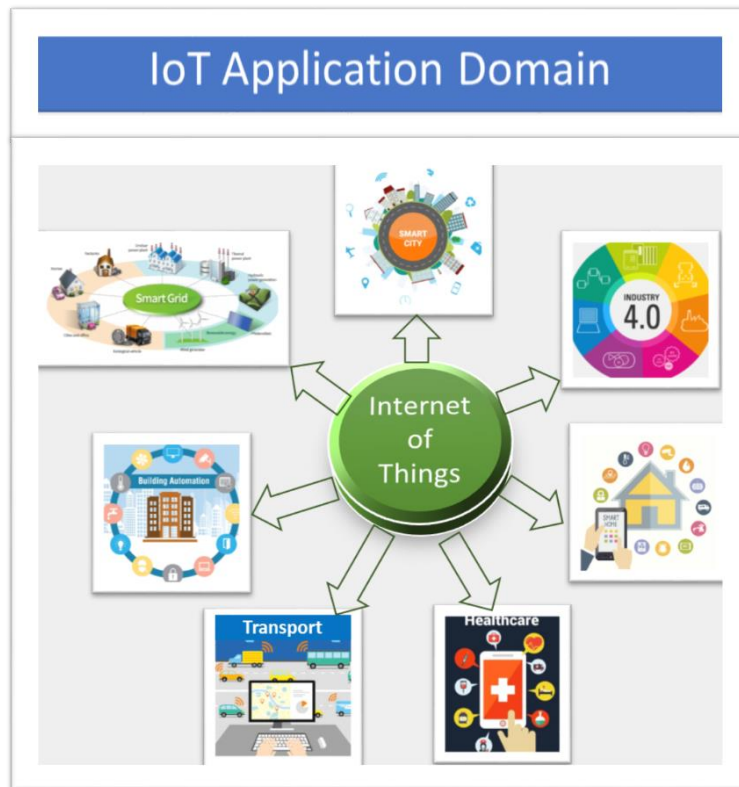


Figure 1-1: IoT Application Domain

IoT applications are becoming essential components of almost every sector (Figure 1-1). Smart Grid is one application area of IoT. The traditional electric

grid interconnects power generation plants to substations and individual households. The smart grid incorporates sensors and actuators to enhance its efficiency, performance and reliability. It also enables new services that are not possible in the traditional grid [1.3], [1.4]. Industrial IoT [1.5], a.k.a. Industry 4.0, is also gaining extreme popularity in recent years. Nowadays, industries are moving into wireless infrastructure that makes extensive use of sensors and actuators. One of the driving forces is the advent of wireless technologies that are specifically designed for industries by taking the stringent performance and reliability requirements of industrial environments [1.6]–[1.8]. Cities are also building smart object networks that span across the entire city to take advantage of IoT services. The objective of these networks is to build IoT applications and services that will improve the quality of lives of their dwellers [1.9]. The IoT services that can be provided are numerous, but ecosystem monitoring and natural hazards monitoring and early detection [1.10] are of paramount importance. Building and home automation are other important application areas of IoT. Modern buildings are being fitted with smart object networks and associated IoT applications that monitor and control various aspects of the building including energy efficiency and security [1.11]. Individual households are also incorporating gadgets and household utensils that can be accessed from the Internet making their home smart, secure and convenient to live in [1.12]. Healthcare [1.13] and transportation management [1.1],[1.10] are also application areas of IoT.

1.2 IoT Systems

A generic IoT system is composed of several components interconnected through different technologies. The intelligence of IoT applications can also reside anywhere across these components. In this section, we discuss the different components of generic IoT systems, communication and connectivity, as well as the placement of the processing logic (Figure 1-2).

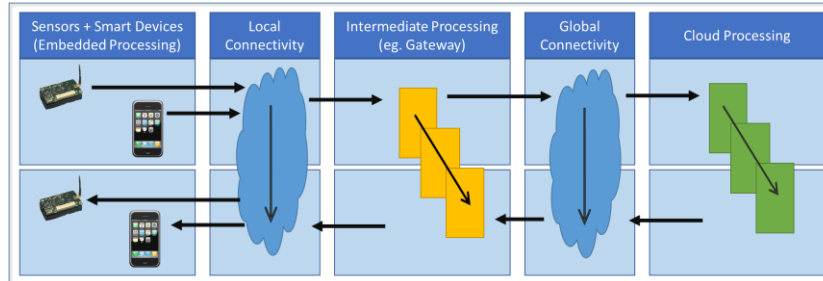


Figure 1-2: Generic IoT System

1.2.1 Components of Generic IoT Systems

In most IoT systems, we can identify three different components that work together.

1. **Constrained Devices** – constrained devices, also called *Smart Objects*, are usually used to interact with the physical world. These devices are sensors and/or actuators fitted with processing and communication hardware. The communication method usually used is wireless. Sensors read some physical value, perform processing, if needed, and communicate the values to interested parties, while actuators alter the physical world based on some events that can be generated internally or communicated from other devices. The smart objects used in IoT applications have severe constraints as compared to other electronic devices such as computers [1.14]. Low processing power, small memory capacity, limited communication power and limited battery power are some of the limitations of these devices. In many situations, these devices are expected to be interconnected with each other, forming Low Power and Lossy Network (LLN), to provide the required information for IoT applications. LLNs are inherently unreliable due to the limitations of the smart objects. Links can break, leading to timeouts or failure of the network in case a new route cannot be established in time. In order to save battery power, nodes (smart object) have to spend most of their time sleeping [1.14]. In addition, some nodes may die due to power drainage, which is also another issue of the LLNs. Due to these facts, developing IoT applications and communication protocols that involve these constrained devices needs special consideration.
2. **LLN Gateways** – LLN gateways are non-constrained devices that sit at the edge of the constrained network to link the LLN with the outside world. The LLN Gateways may have numerous purposes. Some manufacturers use proprietary communication protocols for data communication within the LLN. In such cases, any external device that tries to communicate with the nodes in the LLN from the Internet has to communicate with the gateway using standard Internet communication protocols. Next, the gateway will

enquire the information from the nodes and respond on their behalf. Another purpose is proxying. The gateway may act as a reverse proxy for web-based communication. If the node is not available at the time of the request or does not have the values ready, the gateway may respond on its behalf [1.15]. Gateways may also play a role in security. Since standard security protocols and key management methods are too cumbersome for constrained devices, the gateway can be used as a trusted broker for all security related transactions[1.16]. Some IoT applications also use the gateway as a data collection and analysis point while others use it as intermediary, for instance, between sensors and actuators. This means a sensor notifies an event to the gateway and the gateway sends a trigger to the associated actuator.

3. **The Cloud** – the Cloud has brought several new possibilities to the Internet. Considering the IoT vision, which is connecting things irrespective of their location, capacity or technology they use, the Cloud is an important option to host at least a portion of IoT applications and store generated data. Some IoT applications send all data all the way to the Cloud for processing and analyzing. In the other extreme, only the final result will be sent to the cloud for storage and further analysis. In this case, the constrained nodes, the LLN gateway and/or other intermediary devices perform the bulk of the processing. In any case, the Cloud is becoming an integral component of current IoT applications.

1.2.2 Connectivity and communication

When we closely look at the connectivity of the different components, the LLN network shows peculiar characteristics. Figure 1-3 shows a protocol stack designed to address the specific requirements of constrained devices and networks.

	<i>IETF IoT Protocol Stack</i>	<i>TCP/IP Protocol Stack</i>
Application Layer	IETF COAP	HTTP, FTP, DNS, SSH, SMTP, NTP, ...
Transport Layer	UDP	TCP, UDP
Network Layer	IPv6, IETF RPL	IPv4, IPv6
Adaption Layer	IETF 6LoWPAN	N/A
MAC Layer	IEEE 802.15.4 MAC	Network Access
Physical Layer	IEEE 802.15.4 PHY	

Figure 1-3: IETF Protocol Stack

The most common communication protocol at the lower layer of the protocol stack is IEEE 802.15.4 [1.18]. The protocol defines how to encapsulate, process and

transmit packets to neighboring devices. For multi-hop networks, various routing protocols have been defined [1.19], [1.20], [1.21]. One of the standardized route-over protocols commonly used in LLNs is the Routing Protocol for Low-power and Lossy Network (RPL) [1.19]. RPL operates by establishing a Destination Oriented Directed Acyclic Graph (DODAG) rooted at a particular node. The graph is built by using a fixed metric (e.g. shortest path) to compute the best route. Once the DODAG has been established packets can be routed from nodes to the root or downwards from the root to the nodes. Node to node communication is also supported by RPL. The border router, which is usually connected to the LLN gateway, is typically made the root of the DODAG, so that this node acts as the main router to and from the LLN.

The advent of various standards brings the IoT vision to reality by allowing direct access of sensors and actuators from the Internet. The 6LoWPAN Adaptation Layer [1.22] provides a mechanism for compressing IPv6 packets so that they can be forwarded through the LLN. This is achieved by omitting or compressing header fields to significantly reduce the IPv6 header size in order to fit the packet size restriction imposed by IEEE 802.15.4. The adaptation layer, which sits between the network layer and the data link layer of the OSI model, enables a packet originating from the Internet to directly reach a particular node in the LLN and vice versa.

Another interesting standardization effort can be found at the application layer. Web-services, especially those following the REpresentational State Transfer (REST) paradigm [1.10], are well suited for applications involving smart objects. The REST paradigm allows easy and standardized communication through transfer of states between clients and servers. HTTP is a well known example of a RESTful protocol. However, HTTP is too heavy to be used in IoT applications. Due to this, the IETF has introduced Constrained Application Protocol (CoAP) [1.15] as a light-weight counterpart of HTTP. Like HTTP, CoAP exposes web resource using Uniform Resource Identifiers (URIs). Resource representations are accessed using GET, PUT, POST and DELETE methods. But unlike HTTP, it uses UDP at the transport layer, rather than TCP. For reliable communication, CoAP introduces confirmable messages that require the receiver of a request to send an acknowledgement. An important concept that leverages on CoAP and that is important for IoT applications is the discovery of resources. CoAP specifies resource discovery to be an integral part of the protocol implementation. The `/well-known/core` resource is an entry point to discover all resources hosted on a particular node. This makes it easier for applications to automatically look for specific resources and interact with them without human intervention.

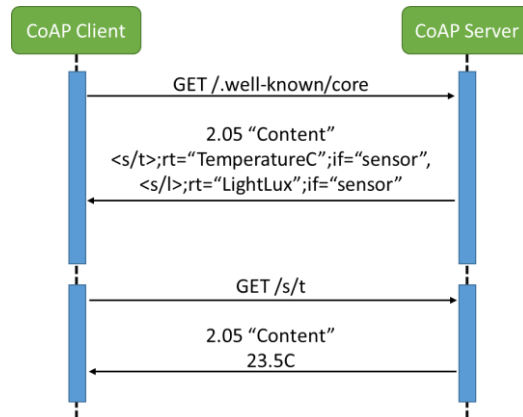


Figure 1-4 CoAP Client-Server Interactions

Figure 1-4 shows a typical interaction between a CoAP client (e.g., smartphone) and server (e.g., temperature sensor). The clients use the node's IPv6 address to send a GET request to the `/.well-known/core` resource in order to get a list of resources hosted on the server. The server responds with the list of resources. This list reveals, amongst others, that the CoAP server exposes temperature sensor readings by means of a resource with the URI path `/s/t`. Next, the client sends a GET request to this resource to request the current temperature, being 23.5°C.

In addition, several extensions have been proposed to CoAP, one of them being the observation of resources [1.23]. The observe extension lets clients inform servers that they want to be notified about every resource state change. Once registered, the server sends a notification every time the state of that specific resource changes. This mechanism is a very important extension for monitoring applications.

1.2.3 Data Processing

In addition to the transfer of data between the different IoT application components using the aforementioned protocols, data processing is also important. One approach is to place the processing logic inside the LLN, possibly distributed over multiple nodes, and perform all processing activity there. This way, the data generated by the constrained nodes is processed and consumed within the LLN. The other extreme are cloud-centric applications where the entire processing logic of the application is performed in the cloud. Each and every data generated in the LLN crosses the network and goes all the way to the cloud [1.17]. Yet another approach is placing the bulk of the processing activity at the gateway. In such applications, every data generated by the constrained devices is sent to the gateway

so that it is processed and analyzed. Depending on the result, data may have to go into the LLN to trigger actuators. For instance, in an IoT application that regulates home temperature, every temperature sensor sends its readings to the gateway for processing. After performing all the required processing, the gateway may send back an actuator message to increase or decrease the temperature. It is also possible to use a hybrid approach where the processing logic of the IoT application can be distributed all over the different components.

1.3 Application development paradigms

IoT is so complex as it encompasses devices with heterogeneous capabilities connected through multiple technologies across numerous domains. Due to the trade-off between processing and data transmission, which can have significant impact on constrained devices, placement of the bulk of the application processing logic plays an important role in IoT application development. As a result, researchers have proposed different architectures based on this fact. The architectures can be categorized as follows.

1.3.1 Cloud-Centric Architecture

Some IoT applications move all the processing logic in the Cloud [1.17],[1.24]. In such architectures, the constrained devices are tasked with communicating sensed data and receiving triggers. For instance, in Light Weight Machine to Machine (LWM2M), the server which sits in the cloud uses the CoAP protocol to perform full device management functions of multiple devices. In such cases, the IoT applications can be built by using APIs on top of the cloud. Referring to Figure 1-2 above, application development paradigms that send the data all the way to the cloud and results coming back to the LLN fall in this category.

1.3.2 Gateway Centric Architecture

It is common to use the gateway as the IoT application processing center. Such architectures let constrained devices send every data to the gateway for processing. Since the gateway sits at the edge of the LLN network, application developers tend to use proprietary protocols for communication between the constrained devices and the gateway. Open standards, such as CoAP can also be used in such application paradigms [1.25][1.26].

1.3.3 Distributed Architecture

A mix of the abovementioned two architectures is also possible. In such architectures, even the constrained devices can host the IoT application logic. An IoT application that requires the constrained devices to perform data aggregation (e.g. averaging) before forwarding to the gateway or which performs more

complex processing and forwards the result to the cloud for storage and further processing are good examples of such an architecture.

1.4 Challenges in Building IoT Applications

Due to the constraints of IoT devices and the networks they are part of, there are several trade-offs that need to be taken into account when developing applications as well as challenges to overcome. In this section, we describe such trade-offs as well as the challenges that are being addressed in this PhD work.

1.4.1 Collection of Redundant Data

In resource monitoring applications, it is very common to collect all sensed data in the Cloud and then decide whether to use it or discard it.

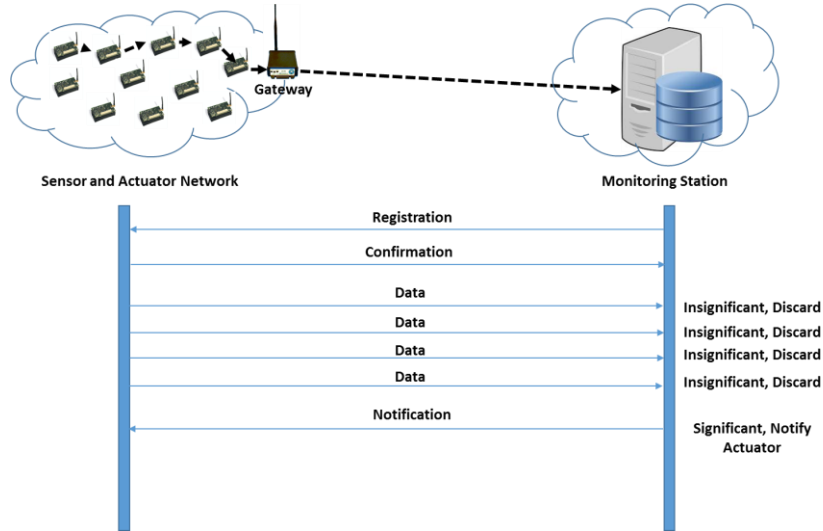


Figure 1-5: Collection of Redundant Data

As an example, we consider a CoAP-based air quality monitoring application (Figure 1-5). In such an application, the data collected by the sensors can be sent to the monitoring station hosted in the Cloud. To do so, the monitoring station establishes an observation relationship with the sensors in order to be notified of every state change. Accordingly, the sensors send every state change to the monitoring station. However, since not every state change is important for the application, the monitoring station discards the insignificant data values. This is very inefficient when we look at it from the constrained device and network perspective. Data reception and transmission are the most energy consuming

operations of the constrained nodes. Sending data all the way to the cloud and discarding it, is a waste of energy and wastes the already scarce bandwidth. In addition, the extra power consumption leads to a reduced lifetime of the nodes. Moreover, in dynamic environments where insignificant changes are common, there will be lots of communication, putting pressure on the nodes near the sink.

A standardized way to collect only the data that is significant from the application perspective is important. The best option would be to filter the data at the server. This server side filtering allows the sink to only receive packets that are significant for the application. This can be achieved by informing the server about the notification criteria while registering for observation.

1.4.2 All Intelligence on Non-Constrained Device or in the Cloud

In most applications, the role of sensors is limited to sensing data and transmitting this data to a more powerful device or to the cloud, while actuators are tasked with receiving information from external devices and altering their environment. This means that all intelligence resides in the cloud or on a more powerful device such as the gateway. This approach can introduce significant delays in the communication and has a huge overhead in terms of power consumption.

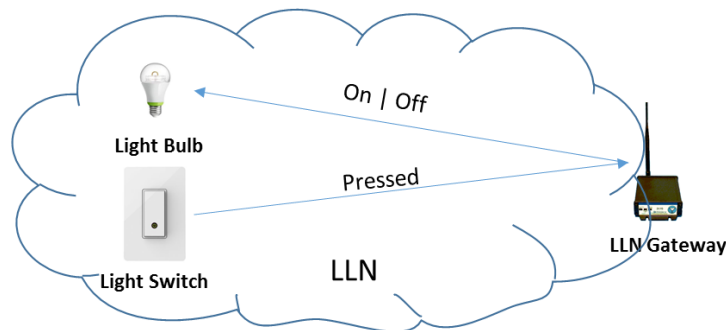


Figure 1-6: Sensor (Light Switch) and Actuator (Light Bulb) Interaction through an Intermediary

For instance, in a home automation system, we may have a light switch and a light bulb that work together by communicating through wireless signals (Figure 1-6). When the switch (Sensor) is pressed, it sends a wireless signal to the light bulb (Actuator) to turn on or off the light. In most cases, the switch press signal is sent to a more powerful device outside the LLN which processes the signal and sends a trigger to the right actuator, introducing significant delays due to the nature of the LLNs. Therefore, if there are application restrictions (e.g. timeouts) set by

applications, there can be communication breakdowns or malfunctioning of the IoT application, especially during peak hours, due to excessive delays.

Solutions that allow direct interaction between constrained nodes can mitigate such issues. This may necessitate to move some of the intelligence into the LLN. However, tasking a constrained node with processing of complex data might be unrealistic. Yet a better solution is to provide a mechanism that enables distributed data processing, relieving the burden for a single node of doing all the processing task. This way, all nodes and the LLN gateway may share the processing activity, thereby reducing both latency and overhead involved in indirect communication. If needed, some processing can still be done in the Cloud. The direct interaction between constrained nodes coupled with distributed processing possibility can be a great enabler for IoT application development.

1.4.3 Static Configurations

CoAP resources are often defined inside the firmware of constrained devices at compile time. However, needs may vary over time. For instance, different CoAP resources that expose data in a different way or with different logic behind them can be required after the nodes are put in production. In addition, updates are required to implement better performance. A solution that allows dynamic deployment of resources and processing logic is imperative. Such solutions add a great deal of flexibility to IoT applications.

1.4.4 Unexpected Crashes of Constrained Devices

Interactions between constrained nodes and external devices usually result in new states that may change over time. Examples of such changes include actuation thresholds set by a device using CoAP PUT request, observation relationship established between a sensor and a monitoring station, or dynamically loaded modules. Unfortunately, constrained devices are characterized by unexpected failures and sometimes they may be put offline for maintenance (e.g. battery change) which leads to loss of the dynamically created states upon startup.

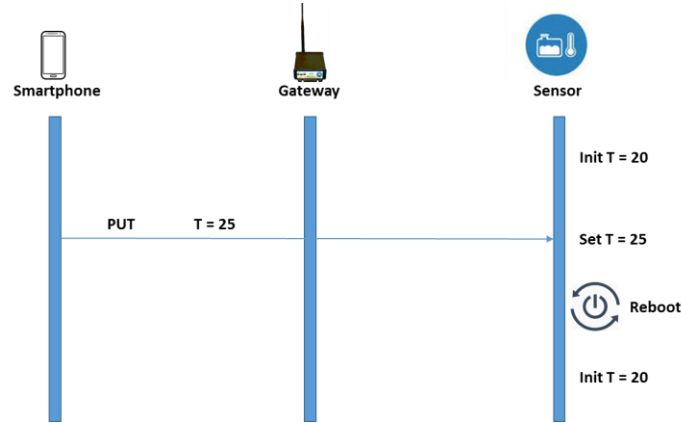


Figure 1-7: Effect of Rebooting Nodes on IoT Applications

Figure 1-7 shows an IoT application which allows smartphones to adjust temperature values of an airconditioning system. The initial configuration (which was 20) can be updated by a user from outside the LLN. If not stored in persistent memory, the value is set back to the initial value upon a reboot. This loss affects all devices that rely on the established dynamic states and relationships. Most importantly, it affects the IoT application that makes use of data and states generated at the rebooted node. A mechanism that keeps track of such dynamic states and that restores them after the reboot has been completed is crucial. Ideally, such a mechanism is fully transparent to both parties (the client and the server). It should also be done by a device that is powerful enough to do the required processing.

1.5 Research Contributions

As mentioned before, the bulk of the PhD work focusses on the usage and extension of the CoAP protocol to improve efficiency and reliability of CoAP-based IoT applications.

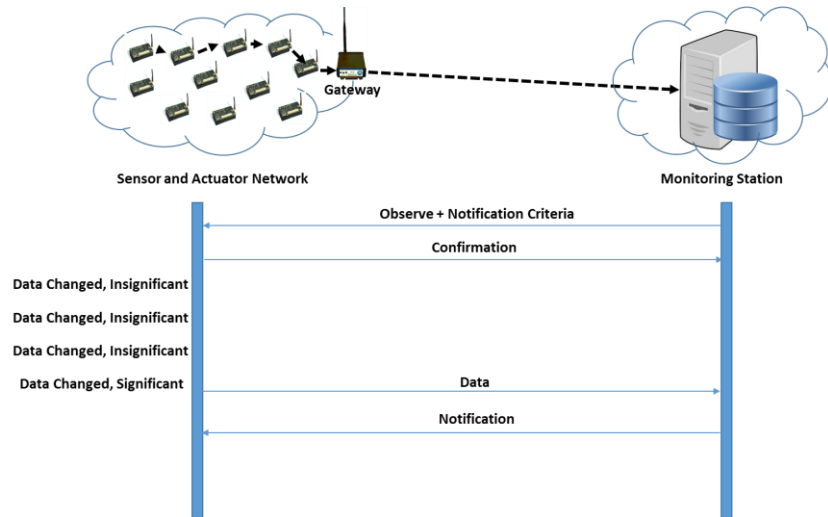


Figure 1-8: Contribution: Conditional Observe

The first contribution of the PhD work is *Conditional Observation* (Figure 1-8). Conditional Observation extends the CoAP Observe protocol to allow notification criteria to be sent along with a registration request as a payload. Upon receipt, the server stores the notification criteria together with all other attributes that are stored for normal observation. Whenever the state of the resource changes, the sensor checks if the new value meets the notification criteria before sending it to the client. This mechanism avoids unnecessary packets that are transmitted to clients. To evaluate the performance of the conditional observe solution against normal observe, we implemented the solution using Contiki. The default CoAP implementation of Contiki, Erbium, was modified to support conditional observation. The hardware platform used is Zolertia Z1 while the Cooja simulator is used to run the actual tests. Theoretical analysis is made to back the results of the experiments. We found out that conditional observe solution results in reduction of traffic load and energy consumption with limited implementation overhead.

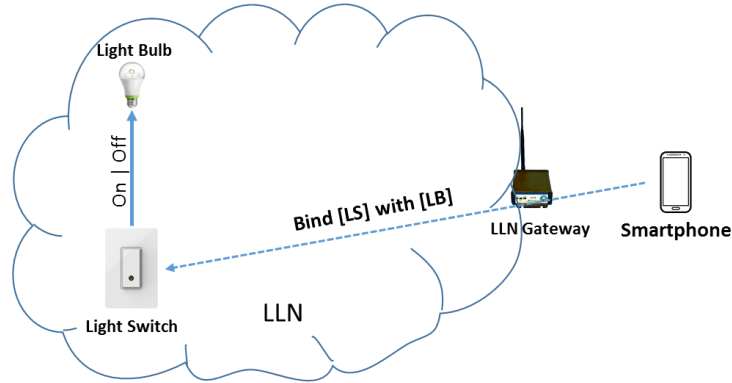


Figure 1-9: Contribution: Bindings (LS=Light Switch and LB = Light Bulb)

The second contribution of the PhD Work is a new enabler to establish direct flexible interactions between two devices, called *Bindings* (Figure 1-9). Bindings are observation relationships between two parties established by a third party (e.g. an observe relationship established between a sensor and an actuator created by using a smartphone). Once the relationship is established the two parties can communicate with each other without the intervention of the relationship creator. We implemented the solution using similar tools (Erbium on Contiki) and used the Cooja simulator to collect data for analysis. The results showed that despite the small increase in memory footprint, the proposed solution meaningfully reduces the flow of packets to the gateway and hence lowers communication delay. Moreover, the new approach makes IoT applications that require direct interaction between sensors and actuators easier and more flexible.

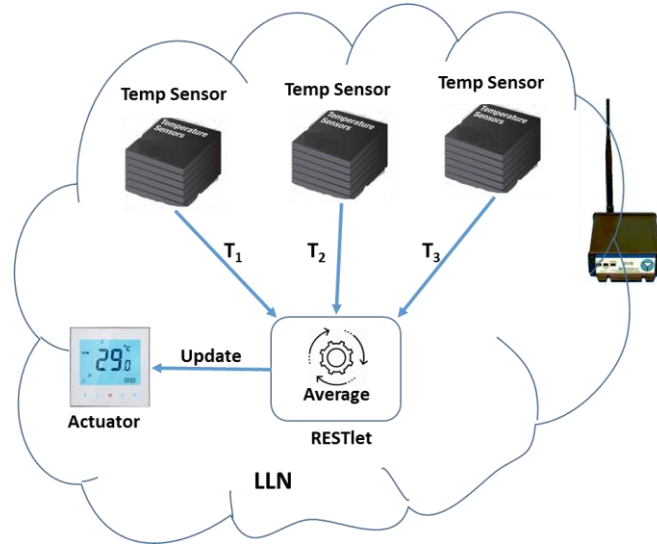


Figure 1-10: Contribution: RESTlets

The research on the binding concept has led to a second novel enabler, called *RESTlets* (Figure 1-10). RESTlets are IoT application building blocks that receive input from sensors or other RESTlets, process them and produce outputs. We can attach readings of multiple sensors to a RESTlet that does averaging and produces the average value as an output. The output, in turn, can be used to trigger an actuator, effectively creating an IoT application that alters the environment by using the average of sensor values. The interconnection between RESTlet inputs/outputs with sensors, actuators and other RESTlets is realized through Bindings. The implementation of the solution on both constrained and non-constrained devices proved that the RESTlet concept can be applied at any location. We ran several experiments to evaluate the performance of our solution by comparing it to traditional gateway-based or cloud solutions by using a different number of data generating nodes, data generating gap and TX/RX ratio. In all cases, our solution is capable of outperforming traditional solutions in terms of latency. Interestingly, the RESTlet solution provides a very good opportunity to use visual programming techniques to reduce the IoT application development to a set of drag-and-drop or point-and-click activities.

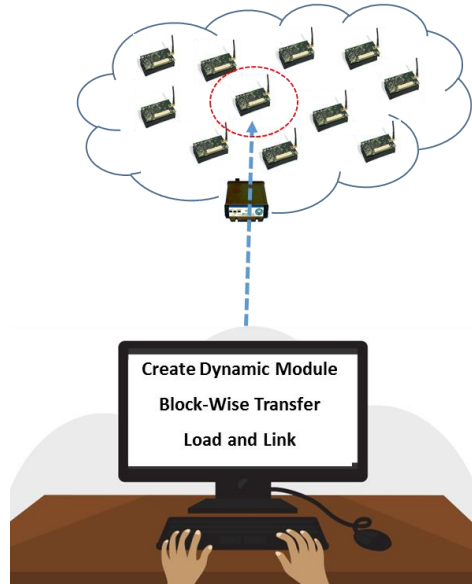


Figure 1-11: Contribution: Dynamic Loading

Initially, we only considered static RESTlets that had to be programmed in advance on the constrained devices. To get rid of this limitation, we investigated the Dynamic Deployment of RESTlets (Figure 1-11). Instead of using statically configured RESTlets, we save lots of memory space by deploying them on demand. The proposed solution involves a way of creating and deploying dynamic RESTlets at run-time. This will increase flexibility of IoT application development, as which node has to host which RESTlet might not be known at compile time. In addition, the RESTlet hosted on a specific node may also have to change over time. Both aspects can be addressed by our method that allows changes on the fly.

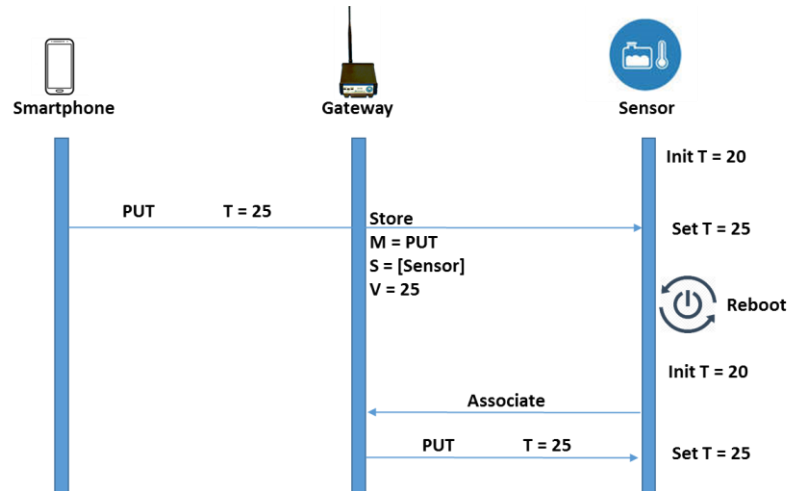


Figure 1-12: Contribution: Crash Recovery

The last contribution of the PhD work is a smart transparent mechanism for crash recovery (Figure 1-12). The solution proposes a dynamic state directory to be placed at the LLN gateway that intercepts each and every packet passing by and that looks for CoAP requests that may generate or alter dynamic state. If a node reports a reboot, the gateway consults the state directory and initiates a sequence of activities that will regenerate the stored dynamic states. This solution ensures continuous functioning of the IoT application despite the rebooting of specific nodes. More interestingly, the solution works without the knowledge and participation of the client and server nodes.

1.6 Fit within the Broader IoT Landscape

Due to the competing requirements of different IoT application domains, various technologies and solutions exist today resulting in a complex landscape. Some companies use their own proprietary solution while others join hands to establish alliances that create standardized solutions. Our work has been centered around the Constrained Application Protocol and has focused on constrained devices that are part of a multi-hop network based on the IEEE 802.15.4 radio technology. In this section, we give a brief overview of the current IoT landscape by referring to the layered architecture given on Figure 1-3 and discuss how our work fits within this broader context.

At the physical and MAC layers, there are numerous communication technologies that target low power devices. For the sake of discussion, we can categorize them as single-hop and multi-hop technologies. An always-on non-constrained gateway

or access point manages and synchronizes communications in single-hop technologies whereas devices may act both as end nodes and routers in multi-hop technologies.

Recently, novel single-hop technologies have appeared, which are aimed at achieving long coverage area at the cost of throughput. These networks are referenced as Low Power Wide Area Networks (LPWANs). One example of such a long-range low-power technology is SigFox. The technology connects low-energy devices such as electricity meters using Ultra-Narrow Band (UBN) technology. The SigFox radio link uses unlicensed ISM radio bands at sub Giga Hertz frequency bands (868MHz in EU and 915MHz in the US) with an average coverage area of 30-50Km in rural areas and 3 – 10Km in urban areas with many obstacles. Because of the duty cycling restrictions of the frequency bands used by SigFox, the number of uplink (device to gateway) messages per day is restricted to 140 with 12 bytes payload per message transmitted at 100bits per second. Smart parking, smart meters, and environment monitoring are target markets of SigFox. Due to its limitation on the transmitted packets, this solution is suitable for solutions that have low data flow requirements. The main contender of SigFox is LoRa, which, together with LoRaWAN, provides a long-range communication solution and corresponding architecture. LoRa uses the same frequency bands as SigFox but supports use of variable bandwidths ranging from 7.8 kHz to 500 kHz allowing different data rates and communication ranges. LoRaWAN solutions can be operator managed or private while SigFox is entirely a service offered by the operator. Typically, proprietary payloads are being exchanged over these technologies, without using IP. However, at the time of writing, the IETF is making efforts to also adopt the same open standards that have been studied in this PhD book by designing novel compression schemes. With this, our contribution to avoid the collection of redundant data becomes particularly relevant to further reduce energy consumption.

Today, long-range IoT connectivity can also be achieved using cellular technology in the licensed spectrum. For this, new cellular specifications have been released, as LTE has been designed to provide high throughput, ending up to be too power hungry for IoT solutions. Considering this, 3GPP has come up with three different LTE based LPWAN standards, namely LTE-M, NB-IoT and EC-GSM-IoT. The services that use these technologies are provided by mobile operators. These technologies have higher peak data rates than LoRa and SigFox, making easier to adopt the open protocol stack considered in this PhD book.

Apart from that, there are also medium and shorter-range technologies that are suited for IoT applications. Current Wi-Fi standards can also be adapted to be used for IoT solutions as the existing standards are not applicable since they use

crowded frequency bands (2.4GHz and 5GHz) with limited range and do not have power saving mechanisms. IEEE 802.11ah (aka Wi-Fi HaLow) is a new Wi-Fi standard that operates in the unlicensed sub-gigahertz frequency bands like SigFox and LoRa but uses wider bandwidths, usually 1 MHz and 2 MHz, in order to achieve larger data rate. Due to this fact, the coverage area of IEEE 802.11ah is much less than that of SigFox and LoRa, but open standards such as IPv6 and CoAP can be easily adopted.

The other widely known single-hop technology is Bluetooth Low Energy (BLE). BLE is built upon the Bluetooth and is usually used with short range sensors such as heart rate belt and weight scale. *Bluetooth Smart* devices implement the BLE standard and *Bluetooth Smart Ready* devices implement both the traditional Bluetooth standard and BLE. BLE devices operate in the 2.4GHz frequency band and coverage is limited to 20 – 50 meters. Recently, a BLE meshing specification has been released, enabling multi-hop communication. BLE specifies almost the entire stack and does not make use of IP(v6) or the CoAP protocol.

When looking at multi-hop communication solutions, it can be observed that the most commonly used radio technology for multi-hop sensor networks is IEEE 802.15.4. IEEE 802.15.4 uses 2.4GHz (worldwide), 868 MHz (EU) and 915 MHz (US) frequency bands with 250kbps, 20kbps and 40kbps data rates, respectively. Since the coverage area is very limited (5 – 50 m) and uses power saving radio and MAC mechanisms, the battery life of nodes may range from months to years. In order to further optimize it, IEEE 802.15.4e has been released. This amendment uses the same physical layer but adds a new MAC layer to introduce Time Slotted Channel Hopping (TSCH) mode. In TSCH mode, the entire frequency range is divided into smaller frequency bands and each sender/receiver node will be assigned a particular time slot and frequency band for communication. In all other time slots, the sender and/or receiver are allowed to sleep to save energy. The assignment of a frequency band and time slot is actually done by an application-aware scheduling mechanism.

Several IoT solutions have been designed on top of IEEE 802.15.4. For instance, ZigBee is a well-known standard for home automation based on IEEE 802.15.4. ZigBee defines a complete protocol stack on top of IEEE 802.15.4 with different profiles addressing different application domains. Several ZigBee based home automation products are available in the market. WirelessHart is another wireless sensor networking technology based on IEEE 802.15.4. It is suitable for process automation in industrial applications. It uses a time synchronized, self-organizing and self-healing mesh architecture. These IoT solutions provide their own standardized stack, not using the open network communication protocol stack (6LoWPAN, RPL, CoAP) that has been defined by the IETF in order to make

constrained nodes directly accessible from IPv6 networks. Thread partially adopts this open stack up to the transport layer, by building on top of 6LoWPAN and UDP. It is backed by Nest Labs (Google) and used in products such as the NEST learning thermostats and related NEST products.

Also at the application layer, a wide variety of approaches exist, one of them being CoAP. Client/Server architectures and Publish/Subscribe architectures are commonly used with IoT applications. Most of the Client/Server applications follow the Resource State Transfer (REST) approach where servers expose data as resources that can be accessed by clients using standardized methods. HTTP and CoAP are protocols that enable RESTful interactions. In the Publish/Subscribe approach, devices publish their data on a particular topic on a broker and subscribers consume them. The publisher and subscriber should not be synchronized or be online at the same time. A notable protocol that is based on publish/subscribe paradigm is the ISO certified Message Queue Telemetry Transport (MQTT) protocol, although CoAP can be used as well to implement this paradigm. In that respect, the IETF is working on a draft to standardize a publish/subscribe broker for CoAP.

To conclude, a plethora of IoT applications, products and solutions that are based on different, often not-interoperable, technologies are currently available. However, there is a tendency to move toward open Internet and Web technologies, where protocols such as IPv6, 6LoWPAN and CoAP take up a dominant role. For instance, the Open Connectivity Foundation (OCF) and Open Mobile Alliance working group for Light-Weight Machine-to-Machine Communication (OMA LWM2M) and IPSO Alliance have selected CoAP as the application protocol on top of which management APIs and data models are being defined. Also, OneM2M has released specifications that include a mapping to CoAP. This illustrates the efforts on creating standards and interoperability in order to reduce the complexity of the existing IoT landscape.

1.7 Outline

Smart objects that are being used in IoT applications have inherent constraints. They have limited memory and processing power. Their communication capabilities are also seriously limited. Moreover, they are battery operated and need power saving mechanisms to extend their battery life. These characteristics make it impossible for existing Internet communication protocols to be directly used by the smart objects. This PhD dissertation is composed of different publications that were realized within the scope of this PhD study. All publications focus on using standardized solutions to provide additional features and give more

flexibility to the IoT. More specifically, all of the PhD work is centered on CoAP and its extension.

The existing CoAP Observe mechanism allows clients to register their interest in state changes of CoAP resources hosted by a server by sending a GET request that includes the observe option. Once registered, the server (usually sensors), notifies the client of every change. However, this is often quite inefficient as not all state changes are relevant for an application, leading to a waste of precious bandwidth and energy. The conditional observation solution we discuss in Chapter 2 enables clients to send notification criteria along with the registration request. By doing so, the server will not send notifications unless they meet the notification criteria.

Chapter 3 introduces two enablers to IoT application development, namely, Bindings and RESTlets. Bindings are CoAP Observe relationships established by a third-party. Bindings enable the creation of direct interactions between sensors and actuators by letting a third party establish an observation relationship between them. RESTlets are IoT application building blocks that receive inputs, process them and produce outputs. They also have control parameters that can be used to tweak configuration parameters. The inputs can be sensor readings or outputs of another RESTlet, while the outputs can be fed into actuators, IoT components at the gateway or in the cloud, or even to other RESTlets as input. The processing logic can be as simple as averaging of inputs or as complex as sending an SMS to a particular destination. RESTlets can be created once and instantiated multiple times in order to perform multiple processing tasks. A RESTlet can be hosted on a constrained node or at non-constrained devices. In addition, we show that IoT application logic can be broken down into smaller units and distributed as RESTlets across the different IoT components. To build IoT applications, each RESTlet input establishes a binding relationship with a resource on sensors or other RESTlet outputs so that each resource state change is reported to the node hosting the RESTlet. Similarly, a RESTlet's output will have an observation relationship with another component. The combination of these two enablers significantly simplifies distributed IoT application development.

One of the drawbacks of the original RESTlets approach is the fact that they are created statically, meaning that they are wasting precious memory when not used. To improve on this, we extend the work to dynamically load RESTlets at run-time. We turn each RESTlet into a dynamically loadable module so that it can be uploaded whenever required. This dynamic loading of RESTlets on constrained devices is detailed in Chapter 4. This mechanism avoids pre-loading of RESTlets on nodes and gives the flexibility of adding them on-demand at run-time.

Due to unexpected failures of nodes, dynamically created information may be lost, including state related to the aforementioned mechanisms. Therefore, Chapter 5 discusses a transparent crash recovery mechanism through deep packet inspection. This mechanism allows nodes to maintain their previous states after rebooting. Finally, Chapter 6 gives concluding remarks and the way forward.

The following Table summarizes the link between the chapters and the IoT application challenges it addresses.

SNo	IoT Application Challenge	Chapter
1.	Collection of Redundant Data	Chapter 2
2.	All Intelligence on Non-Constrained Device or in the Cloud	Chapter 3
3.	Static Configurations	Chapter 4
4.	Frequent Crash of Constrained Devices	Chapter 5

1.8 List of publications

The research results obtained as part of this PhD have been published in scientific journals and presented in various international conferences. The following list provides an overview of all publications.

1.8.1 A1 publications (listed in the Science Citation Index¹)

1. **Girum Ketema Teklemariam**, Jeroen Hoebeke, Ingrid Moerman, Piet Demeester. *Facilitating the creation of IoT applications through conditional observations in CoAP*. Published in EURASIP Journal on Wireless Communication and Networking, 2013:177. DOI: 10.1186/1687-1499-2013-177
2. Floris Van den Abeele, Jeroen Hoebeke, **Girum Ketema Teklemariam**, Ingrid Moerman, Piet Demeester. *Sensor Function Virtualization to Support Distributed Intelligence in the Internet of Things*. Published Wireless Personal Communications 81 (4), pp. 1415-1436
3. **Girum Ketema Teklemariam**, Floris Van Den Abeele, Ingrid Moerman, Piet Demeester and Jeroen Hoebeke. *Bindings and RESTlets: A Novel Set of*

¹ The publications listed are recognized as ‘P1 publications’, according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index - Science or Conference Proceedings Citation Index - Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1

CoAP-Based Application Enablers to Build IoT Applications. Sensors 2016, 16(8), 1217; doi:10.3390/s16081217

4. **Girum Ketema Teklemariam**, Floris Van den Abeele, Peter Ruckebusch, Ingrid Moerman, Piet Demeester, Jeroen Hoebeke. *Dynamic Deployment of RESTlets on Constrained Devices*. Submitted to International Journal of Distributed Sensor Networks May 2017.
5. **Girum Ketema Teklemariam**, Floris Van den Abeele, Ingrid Moerman, Piet Demeester, Jeroen Hoebeke. *Transparent Recovery of Dynamic States on Constrained Nodes through Deep Packet Inspection*. Submitted to Journal of Sensors, December 2017

1.8.2 Publications in other International Journals

1. Isam Ishaq, David Carels, **Girum Ketema Teklemariam**, Jeroen Hoebeke, Floris Van den Abeele, Eli DePoorter, Ingrid Moerman, and Piet Demeester. *IETF standardization in the field of the Internet of Things (IoT): a survey*. Published in the Journal of Sensor and Actuator Networks, Volume 2, issue 2, pp. 235–287, 2013.

1.8.3 Publications in International Conferences (listed in the Science Citation Index²)

1. **Girum Ketema Teklemariam**, Jeroen Hoebeke, Ingrid Moerman, Piet Demeester, Li Shi Tao, Antonio J. Jara. *Efficiently Observing Internet of Things Resources*. Published in the proceedings of 2012 IEEE International Conference on Green Computing and Communications (GreenCom 2012), 20–23 Nov. 2012 Pages 446 – 449. Besançon, France
2. **Girum Ketema Teklemariam**, Jeroen Hoebeke, Floris Van den Abeele, Ingrid Moerman, Piet Demeester. *Simple RESTful Sensor Application Development Model Using CoAP*. Published in the Proceedings of the Conference on Local Computer Networks, LCN 2014, November 2014, 6927702, pp. 552-556. Edmonton, Canada.

1.8.4 Publications in international conferences

1. **Girum Ketema Teklemariam**, Jeroen Hoebeke, Ingrid Moerman, Piet Demeester, *Flexible, direct interactions between CoAP-enabled IoT devices*. Published in the Proceedings of 8th International Conference on

² The publications listed are recognized as ‘P1 publications’, according to the following definition used by Ghent University: P1 publications are proceedings listed in the Conference Proceedings Citation Index - Science or Conference Proceedings Citation Index - Social Science and Humanities of the ISI Web of Science, restricted to contributions listed as article, review, letter, note or proceedings paper, except for publications that are classified as A1.

- Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2014. 6975483, pp. 322-327. Birmingham, United Kingdom
2. Floris Van den Abeele, Jeroen Hoebeke, Isam Ishaq, **Girum Ketema Teklemariam**, Jen Rossey, Ingrid Moerman and Piet Demeester. *Building embedded applications via REST services for the Internet of Things*. Published in the proceedings of the 11th ACM Conference on Embedded Network Sensor Systems (SenSys - 2013), p. 1–2, 11–15 Nov. 2013, Rome, Italy.
 3. Jeroen Hoebeke, David Carels, Isam Ishaq, **Girum Ketema Teklemariam**, Jen Rossey, Eli Depoorter, Ingrid Moerman, Piet Demeester, *Leveraging upon standards to build the Internet of Things*, Published in Proceedings of the 19th IEEE Symposium on Communications and Vehicular Technology in the Benelux (IEEE SCVT 2012), Eindhoven, November 16, 2012, DOI:10.1109/SCVT.2012.6399412

1.8.5 Patent Applications

1. Floris Van den Abeele, Jeroen Hoebeke, **Girum Ketema Teklemariam**. *Reducing a Number of Server-Client Sessions*. US2016006818. Koninklijke KPN N.V., iMinds VZW, Universiteit Gent. Priority date: 28 December 2012. Publication date: 7 January 2016.
2. Jeroen Hoebeke, **Girum Ketema Teklemariam**, Floris Van den Abeele. *Binding Smart Objects*. US2017017533. Koninklijke KPN N.V., iMinds VZW, Universiteit Gent. Priority date: 23 December 2013. Publication date: 19 January 2017

References

- [1.1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Futur. Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.
- [1.2] ITU-T, "Recommendation ITU-T Y.2060: Overview of the Internet of things." International Telecommunication Union, Geneva, 2013.
- [1.3] P. P. Parikh, M. G. Kanabar, and T. S. Sidhu, "Opportunities and challenges of wireless communication technologies for smart grid applications," *IEEE PES Gen. Meet.*, no. Cc, pp. 1–7, 2010.
- [1.4] M. Yun and B. Yuxin, "Research on the architecture and key technology of Internet of Things (IoT) applied on smart grid," *2010 Int. Conf. Adv. Energy Eng. ICAEE 2010*, pp. 69–72, 2010.
- [1.5] L. Da Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Trans. Ind. Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.
- [1.6] J. Song *et al.*, "WirelessHART: Applying wireless technology in real-time industrial process control," *Proc. IEEE Real-Time Embed. Technol. Appl. Symp. RTAS*, pp. 377–386, 2008.
- [1.7] R. Show, "The ISA100 Standards," *ISA Stand.*, 2008.
- [1.8] S. Petersen and S. Carlsen, "WirelessHART versus ISA100.11a: The format war hits the factory floor," *IEEE Ind. Electron. Mag.*, vol. 5, no. 4, pp. 23–34, 2011.
- [1.9] a Zanella, N. Bui, a Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for Smart Cities," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 22–32, 2014.
- [1.10] J. P. Vasseur and A. Dunkels, *Interconnecting Smart Objects with IP*. 2010.
- [1.11] C. Wei and Y. Li, "Design of energy consumption monitoring and energy-saving management system of intelligent building based on the Internet of things," *2011 Int. Conf. Electron. Commun. Control. ICECC 2011*, pp. 3650–3652, 2011.
- [1.12] M. Soliman, T. Abiodun, T. Hamouda, J. Zhou, and C. H. Lung, "Smart Home: Integrating Internet of Things with Web Services and Cloud Computing," *2013 IEEE 5th Int. Conf. Cloud Comput. Technol. Sci.*, vol. 2, no. November 2015, pp. 317–320, 2013.
- [1.13] S. M. R. Islam, D. Kwak, H. Kabir, M. Hossain, and K.-S. Kwak, "The Internet of Things for Health Care : A Comprehensive Survey," *Access, IEEE*, vol. 3, pp. 678–708, 2015.

-
- [1.14] C. Bormann, M. Ersue, and A. Keranen, “RFC 7228: Terminology for Constrained-Node Networks.” IETF, pp. 1–17, 2014.
 - [1.15] Z. Shelby, K. Hartke, and C. Bormann, “RFC 7252: The Constrained Application Protocol (CoAP).” IETF, pp. 1–112, 2014.
 - [1.16] F. Van Den Abeele, T. Vandewinckele, J. Hoebeke, I. Moerman, and P. Demeester, “Secure communication in IP-based wireless sensor networks via a trusted gateway Secure communication in IP-based wireless sensor networks via a trusted gateway,” no. October, 2015.
 - [1.17] M. Kovatsch, M. Lanter, and S. Duquennoy, “Actinium: A RESTful runtime container for scriptable internet of things applications,” *Proc. 2012 Int. Conf. Internet Things, IOT 2012*, pp. 135–142, 2012.
 - [1.18] IEEE Standards Association, *IEEE 802.15.4-2015 Standard*, vol. 2015. 2015.
 - [1.19] T. Winter, P. Thubert, A. R. Corporation, and R. Kelsey, “RFC6550: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks.” IETF, pp. 1–157, 2012.
 - [1.20] C. Karlof and D. Wagner, “Secure routing in wireless sensor networks: attacks and countermeasures,” *Proc. First IEEE Int. Work. Sens. Netw. Protoc. Appl. 2003.*, pp. 113–127, 2003.
 - [1.21] M. S. G. Premi and K. S. Shaji, “MMS Routing for Wireless Sensor Networks,” *2010 Second Int. Conf. Commun. Softw. Networks*, pp. 482–486, 2010.
 - [1.22] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, “RFC 4944: Transmission of IPv6 Packets over IEEE 802.15.4 Networks.” IETF, pp. 1–30, 2007.
 - [1.23] K. Hartke, “RFC 7641: Observing Resources in the Constrained Application Protocol (CoAP).” IETF, pp. 1–30, 2015.
 - [1.24] Alliance Open Mobile, “Lightweight Machine to Machine Requirements,” pp. 1–112, 2012.
 - [1.25] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, “IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things,” *2010 IEEE/IFIP Int. Conf. Embed. Ubiquitous Comput.*, pp. 347–352, 2010.
 - [1.26] M. Aazam and E. Huh, “Fog Computing and Smart Gateway Based Communication for Cloud of Things,” no. April, pp. 21–24, 2014.

2

Facilitating the creation of IoT applications through conditional observations in CoAP

Many IoT applications involve the monitoring of the physical world by means of sensors. When using the CoAP protocol, such monitoring can be achieved by making use of the Observe mechanism. This mechanism enables a client (e.g., a monitoring station) to express its interest in the state changes of a resource hosted by a server (e.g. sensor). As a result, the client will be notified about changes in the state of that resource. This way, the station avoids constant polling for resource states, significantly reducing the amount of packet transmissions between the client and the server. However, since the server sends every small change in resource states, which might be insignificant, there is still room for optimization. In this chapter, we introduce the conditional observation mechanism, which lets clients specify notification criteria along with the observation request, and assess its efficiency.

Girum Ketema Teklemariam, Jeroen Hoebeke, Ingrid Moerman, Piet Demeester. *Facilitating the creation of IoT applications through conditional observations in CoAP*. Published in EURASIP Journal on Wireless Communication and Networking, 2013:177. DOI: 10.1186/1687-1499-2013-177

Abstract: *With the advent of IPv6, the world was getting ready to incorporate billions of smart objects into the current Internet to realize the idea of the Internet of Things (IoT). However, one of the major challenges was that existing standard protocols and applications had not been designed with the resource constraints of these smart objects in mind. Currently, a number of initiatives are witnessed to resolve this situation both at the networking level and the service level. One such an initiative at the service level is the introduction of the Constrained Application Protocol (CoAP). This protocol is designed to meet the constraints of smart embedded objects and has the ability to easily translate to the prominent REST implementation, HTTP (and vice versa). The protocol has several optional extensions, one of them being, resource observation. With resource observation, a client may ask a server to be notified of every state change of the resource. CoAP, together with its observe functionality, provides the basis for the integration of constrained devices with the Internet at the service level and the realization of embedded web services. However, in order to really facilitate IoT application design, additional CoAP related functionalities are expected to appear. For instance, many applications can benefit from a lightweight solution for subscribing for very specific events. In this paper, we introduce a compact and lightweight CoAP extension, named Conditional Observation that facilitates realizing this behavior easily across all resources by including notification criteria to be specified along with observation request. We demonstrate the feasibility of implementing this on a constrained device and evaluate the resulting performance in detail. Complemented by a theoretical evaluation, we prove that this mechanism offers several benefits when used in constrained networks with varying properties and discuss its relevance in the realization of IoT applications through a number of use cases.*

2.1 Introduction

Remarkable advances in Microelectromechanical systems (MEMS) have led to the creation of tiny but crucial embedded devices such as sensors and actuators. The wireless communication capability of these devices turns them into smart objects that can interact with the virtual world. Coupled with the explosive expansion of wireless and mobile technologies there are very good reasons to consider these objects as corner stones of the Future Internet rather than mere add-ons to the current communication networks. The resulting Internet is now commonly referred to as the Internet of Things (IoT). However, the severe limitations of these smart objects in terms of memory, processing capacity, power and bandwidth pose great challenges in realizing this. A typical smart object may have a few kilo bytes of memory (RAM and ROM), slow micro controllers, and limited bandwidth (around 250kbps). On top of this, most of the smart objects are battery operated and have limited lifetime. The protocols and applications that are widely used in

the current Internet are too heavy for such constrained devices to be applied directly. Several initiatives exist to alleviate these prevailing problems by proposing new light weight protocols suitable for constrained devices and networks. The Internet Engineering Task Force (IETF) is the pioneer in producing standards and protocols that fit the strict requirements of such constrained environments by establishing working groups that address different aspects of the requirements of the constrained objects and networks.

The IPv6 for Low Power and Lossy Wireless Personal Area Network (6LoWPAN) working group of IETF has produced standards that enable IPv6 to be used in the most constrained devices [2.1]. [2.2] introduces the 6LoWPAN Adaptation Layer which resides between the Network and Link layer and provides three basic services: IPv6 header compression, fragmentation and mesh under routing support. These basic services ensure that constrained devices can talk to unmodified IPv6 hosts in the Internet, and the other way around, while the 6LoWPAN Adaption Layer overcomes the differences in protocol design between these two worlds, necessitated by the constraints of the Low Power and Lossy Networks (LLNs). Further, current routing protocols and algorithms are not suitable for constrained environments for several reasons: high resource (memory, processing, and bandwidth) requirement, absence of uniform metric in LLNs and unreliability of intermediate routing nodes. The Routing in Low Power and Lossy Networks (ROLL) working group is tasked with proposing routing solutions suitable for constrained networks and devices. The Routing Protocol for Low-power and Lossy Networks (RPL) is a proposed standard by this working group [2.3].

Both IETF groups have realized the interconnectivity between tiny objects and the current Internet in a standardized way. However, this connectivity is merely an enabler required to unlock all potential of the IoT in the form of novel applications and services. Web service technology made the success of the current Internet. Now it is expected that an embedded counterpart of web service technology is needed in order to exploit all great opportunities offered by the Internet of Things, since existing application layer protocols, such as HTTP, SOAP, and XML are even heavier than the protocols defined in layers below. Therefore, the Constrained RESTful Environments (CoRE) working group was established to specifically work on the standardization of a framework for resource-oriented applications, allowing the realization of RESTful embedded web services in a similar way as traditional web services [2.4]. Their work resulted in the Constrained Application Protocol (CoAP), a specialized RESTful web transfer protocol for use with constrained networks and nodes. It uses the same RESTful principles as HTTP, but it is much lighter so that it can be run on constrained devices [2.4]. In addition, the group designed observe functionality in order to

allow a device to publish a value or event to another device that has subscribed to be notified of changes in the resource representation [2.14].

CoAP, together with its observe functionality, provides the basis for the integration of constrained devices with the Internet at the service level and the realization of embedded web services. However, in order to really facilitate IoT application design, additional CoAP related functionalities are expected to appear. For instance, many applications can benefit from a lightweight solution for subscribing for very specific events. Ideally, this is built into the CoAP protocol as an extension, avoiding the need to implement such functionality on a per resource basis. This facilitates the realization of many sensor–actuator interactions which typically have the following pattern: if “condition fulfilled” then “take action”. The contribution of this paper is that we present an extension of the CoAP observe functionality that exactly facilitates realizing this behavior by including notification criteria to be specified along with observation request. This way the server will not just send notifications whenever the state of a resource changes. It will first check if the change is significant enough for the client by comparing the new value with the notification criteria sent by the client. Only then, a notification will be sent. The design is compact, lightweight and can be easily shared across all resources. Further, we are the first to implement such an extension to CoAP on constrained devices and to evaluate in detail the potential reduction in power consumption and number of packets transmitted that can be achieved, which is of great importance to constrained networks.

Section 2 of the paper first introduces the CoAP protocol, followed by the existing CoAP Observe option, its limitations and possible approach to tackle these limitations. Related work will be discussed in the section 3. In Section 4 an alternative method, called conditional observation, is presented and the approach is explained in great details. The next section discusses our implementation on constrained devices, followed by section 6 presenting a detailed experimental and mathematical evaluation. In section 7 we further illustrate some potential IoT applications that can benefit from our proposal. Finally, the paper draws conclusions and suggests future work.

2.2 The Constrained Application Protocol and Observe

REpresentational State Transfer (REST) uses mechanisms that are less memory and processing power intensive [2.5]. As a result, many systems are now becoming RESTful [2.4]. In this approach data or resources that must be exchanged between client and server are encoded as representations of the resource. In addition, all states required to complete a request must be provided along with the request. The

desired communication result is achieved by transferring the representations and the states between the client and the server using HTTP operations such as GET, PUT, POST and DELETE [2.5]. However, today's web service technology is a poor match for the vast majority of constrained networks, machine-to-machine (M2M) applications and embedded devices because of their overhead and complexity. For applications that involve smart objects, such as industry automation, transport logistics, and building automation, an embedded alternative would be ideal since it is in line with current web services, facilitating the integration of objects into the Internet.

1 Byte			1 Byte	2 Bytes	TKL Bytes	1 Byte	Variable
V	T	TKL	Code	Message ID	Token (if any)	0xFF (if any)	Payload (if any)
2	2	4bits					

Figure 2-1: CoAP Message Format consisting of a 4-bytes base binary header followed by optional extensions

The Constrained Application Protocol (CoAP) is a protocol proposed by this IETF CoRE working group allowing these RESTful web services to be implemented on constrained objects. CoAP provides exactly the subset of HTTP methods (GET, PUT, POST and DELETE) that is necessary to offer RESTful web services in a WSN-compatible manner [2.4]. This implies that a simple mapping between HTTP and CoAP can be realized (and vice versa) in a similar way that 6LoWPAN can be translated into IPv6 and the other way around. The main advantage is that CoAP has a much lower header overhead and parsing complexity than HTTP. It uses a 4-byte base binary header that may be followed by compact binary options and a payload. In addition, CoAP provides optional transport reliability, normally a core functionality of TCP, which is due to the resource constraints by nature not available in Wireless Sensor Networks (WSNs). This is particularly useful, since CoAP is designed to be used in combination with UDP, which does not offer any reliability but is adequate for WSNs due to its low impact on resources. CoAP can run on top of 6LoWPAN networks, but also on top of proprietary networks that are connected to IPv6 Internet. Figure 2-1 shows the CoAP message format as specified in version 13 of the draft [2.4]. The 4-bytes base header consists of the following fields: Version, Type, Token length, Code and Message ID. The 2-bit Type field indicates whether the message is a confirmable, non-confirmable, acknowledgement or reset message. The Code field indicates if the message carries a request (specifying the method: GET, PUT, POST or DELETE), response (specifying the response code) or is empty. The base header may be followed by one or more optional fields. First of all, there is the optional Token field having a length between 0 and 8 bytes. Next, a variable number of options can follow and finally, if there is a payload, a Payload Marker and the Payload complete the message.

4 bits	4 bits	0 - 2 Bytes	0 - 2 Bytes	0 or more Bytes
Option Delta	Option Length	Option Delta (Extended)	Option Length (extended)	Option Value

Figure 2-2: CoAP option format

The format of a single CoAP option is shown in Figure 2-2. To be able to offer communication needs that cannot be satisfied by the base binary header alone, CoAP defines a number of options which can be included in a message. Each option instance in a message specifies the Option Number of the defined CoAP option. Instead of specifying the Option Number directly, the instances must appear in order of their Option Numbers and a delta encoding is used between them. The Option Length indicates the length of the Option Value in bytes and the Option Value is the actual representation of the option (e.g. an unsigned integer, a code representation, etc.). If the delta value or length is larger than 12, 1 or 2 additional bytes are used to represent the delta or the length.

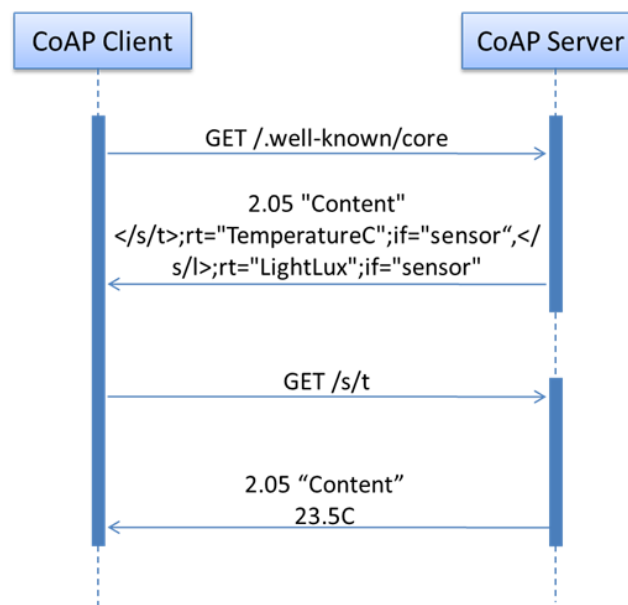


Figure 2-3: CoAP Client/Server communication

Since CoAP is recommended for M2M interaction, automatic resource discovery is made part of the protocol using the CoRE Link Format. A well-known URI, “/.well-known/core”, is defined as an entry point for all links to resources hosted by a server [2.6]. Once the list of resources is identified, clients may send requests

to find out specific values for the resources. As Figure 2-3 depicts, the client first requests the list of resources using GET and the server replies with the list of resources it has. At a later time, the client requests for the current temperature value using another GET, to which the server replies with a response containing the temperature value of 23.5°C. All exchanges use the message format shown in Figure 2-1 and Figure 2-2.

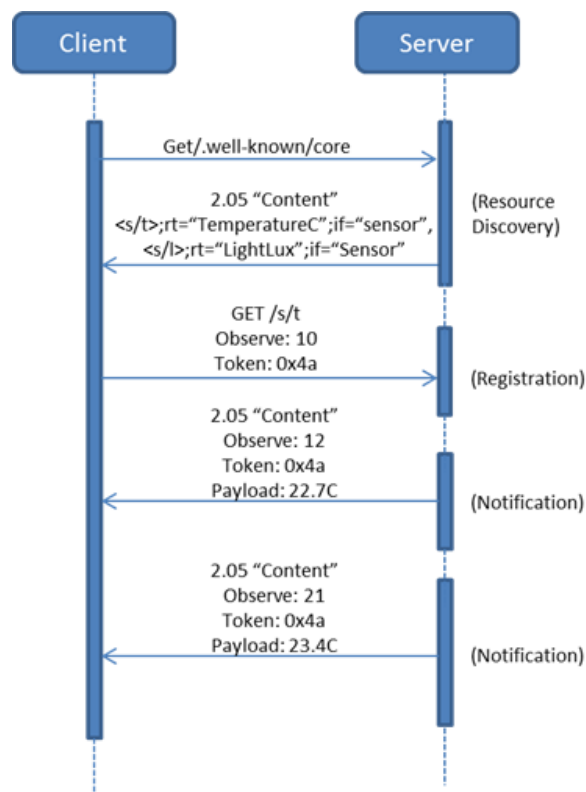


Figure 2-4: Normal Observation

In addition to the main CoAP draft, a number of extensions have been proposed. One of those extensions is the observation of resources through the use of the observe option. The observe option may be used by clients interested to have up-to-date information about the state of a resource as stated in [2.14]. This draft specifies a simple protocol extension to CoAP that gives clients the ability to observe changes of a resource. It uses the well-known observer design pattern,

where clients that are interested in the state of a resource register³ their interest with the server that hosts the resource by sending a CoAP request containing the Observe option. Once registered, clients will receive notifications - CoAP responses containing the Observe option - upon every state change of the resource. In addition, if the state of a resource does not change over time, the server will send a new notification latest after Max-Age of the resource expires. Since the CoAP option Max-Age indicates the freshness of the resource, it is clear that through this observe extension clients will always have a fresh and up-to-date representation of the resource. Figure 2-4 shows how the observe option is used to get up-to-date resource states.

As such, when observe is used, the CoAP client will get a notification response whenever the state of the observed resource changes or its max-age expires. For frequently changing resources or resource with a low Max-Age value, this results in frequent notifications, which is not ideal in constrained networks. Also, it is unclear how non-cacheable (Max-Age equal to 0) resources should be handled. In many cases, an observer will typically be interested in state changes that satisfy a specific condition, instead of receiving all state changes or notifications that only update the freshness.

However, the current observe draft stresses on providing the clients with up-to-date information about the state of a resource. Applications that are interested in values that exceed some thresholds will simply drop the transmitted packets upon reception if they do not meet their criteria (client side filtering). This unnecessary data transmission can be costly to the already constrained objects. The increased number of packet transmissions in highly dynamic environment will also increase the network congestion and for larger networks the impact can be significant. In addition, the power consumption (processing, transmission and listening) can be higher for the overall network.

Therefore, in several cases, one could benefit from a solution for subscribing to very specific events only, i.e. conditional observations. Since we are dealing with constrained devices, such a solution should satisfy several requirements. The functionality should have a sufficiently small footprint, allowing the implementation on very constrained devices. It should be usable by all resources on a constrained device without additional programming complexity. Further, it should offer sufficient expressiveness in order to be able to express conditions that

³ At the time of this work, the Observe draft stated that sending an observe request to the same resource for the second time cancels an observation. But currently, according to the observe RFC (RFC7641), this is changed. Cancellation is done by sending an observe request to the resource with observe value 1.

are encountered frequently across resources and across IoT use cases. Finally, if needed, extensions should be possible in order to cope with future requirements. Based on these requirements, we have chosen to realize this conditional observe functionality by embedding it in the CoAP protocol as a new CoAP option. Before presenting our solution, we will first discuss related work that aims to achieve similar functionality and position it against our approach.

2.3 Related work

There are a number of research activities under way on resource observation in WSNs. Different groups are using different approaches to come up with outstanding solutions and technologies. Publish/Subscribe systems are widely used in the Internet already for a while. The basic concept of such systems is similar to normal observation where subscribers register at publishers (notifiers) and get responses depending on the original request made by the subscribers [2.7]. Different authors have proposed similar solutions to be used in wireless sensor networks. MQTT-S is a protocol proposed to handle pub/sub issues in WSNs. The protocol is based on the MQTT protocol, an established protocol for lightweight publish/subscribe reliable messaging transport, optimized to connect physical world devices/messages and events with enterprise servers and other consumers. The protocol introduces MQTT-S gateways and forwarders to communicate pub/sub information between clients and the MQTT broker, which ultimately responds with the required information [2.8]. With this approach, a 3rd party is required to realize the desired functionality, whereas we want to allow direct end-to-end interactions with the constrained devices. There are also middleware based pub/sub solutions such as Mires [2.9] and PSWare [2.10]. Most of these solutions introduce a new protocol specifically addressing this issue while our approach, however, is an extension of an existing protocol that is being developed in an open standardization organization. This way our approach significantly reduces the additional memory and processing requirement for realizing this new functionality, since it builds upon functionality already present in any CoAP implementation.

The European Telecommunications Standards Institute (ETSI) has also proposed a standard to address observation relationships in Machine-to-Machine (M2M) communications. The ETSI Machine-To-Machine (M2M) Communications functional architecture [2.11] states how RESTful web services can be used in M2M communications. Subscription management is one of the areas the document addresses. In the document, a client may subscribe for a specific resource or an attribute of a resource by specifying filtering criteria, if required. The ETSI standard follows its own functional architecture that is totally different from the

IETF approach. Our solution is based on the work of the IETF CoRE working group.

Another related work is [2.12] where conditional observation requests are represented by URI queries. An important problem with this approach is its complexity. The queries that are generated may have limited readability and could be difficult to represent. Furthermore, URI queries are very resource specific complicating automatic processing of conditional observations or code reuse over several resources. Using a CoAP option for conditional observations makes this functionality independent of any specific resource implementation, whereas URI queries can be used for resource specific functionalities. Further, the link with the Observe option is lost by spreading this functionality over both URI queries and options and the multitude of URI queries that can occur makes it more complex for intermediaries to process this information. Another alternative could be the realization of a new CoAP resource on the constrained device for every event clients are interested in. For example, one could create a resource that has the value 1 when the temperature is smaller than 27 Centigrade and 0 when the temperature is larger. By observing this resource, a client could be informed about the change of this resource and thus the occurrence of this event. It is clear that such an approach is cumbersome and not generic at all. At the same time it puts a serious burden on these constrained devices. Supporting new events implies adding resources, burdening the server and possibly implying flashing the device.

Finally, the Open Geospatial Consortium (OGC) Inc. has been developing different standards in the area of geospatial data. One of the standards developed by the OGC is the Sensor Observation Service (SOS) that deals with the specifications of data observation from different sensors in different, possibly geographically scattered, sensor networks [2.13]. The standard specifies that a GetObservation request may have several mandatory and optional parameters. One of the optional parameters is featureOfInterest, which is similar to our observation type. However, this approach is more focused for geographical observations and is a subset of a bigger framework, which significantly differs from the IETF recommendation.

2.4 Conditional Observe

To avoid transmission of unwanted notifications to clients, the authors of this paper have proposed a new CoAP option “Condition” as an extension to the Observe Option in order to support conditional observations according to the

conditional observe draft [2.15]⁴. This option can be used by a CoAP client to specify the conditions the client is interested in. Now, only when the condition is met, the CoAP server will send a notification response with the latest state change. When the condition is not met, the CoAP server will not send the notification response. Figure 2-5 shows the operation of conditional observation.

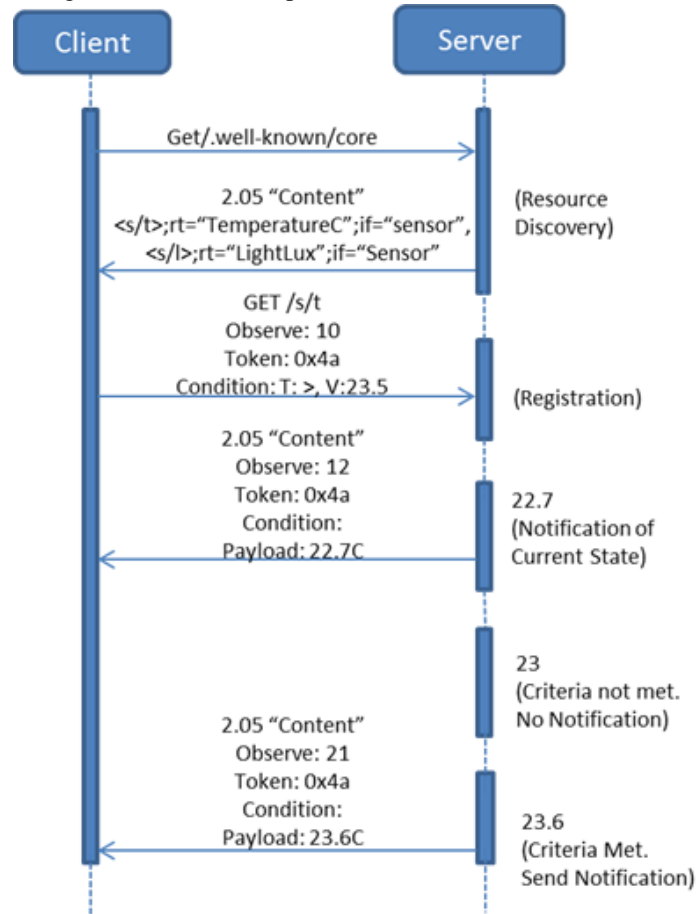


Figure 2-5: Conditional Observation

The Condition option has to be used in combination with the Observe option and can be used both in request and response messages. In a GET request message, the Condition option represents the condition the client wants to apply to the

⁴ The Conditional Observe draft, created a series discussion among the IETF community mainly about how to implement the functionality. Currently, a new internet draft, *Dynamic Resource Linking for Constrained RESTful Environments*, which includes most of the content of the conditional observe draft is now under consideration for approval.

observation relationship. It is used to describe the resource states the client is interested in. In the response to the initial GET request message, the Condition option, together with the Observe option, indicates that the client has been added to the list of observers and that notifications will be sent only when the resource state meets the condition specified in the Condition option. In all further notifications, the Condition option identifies the condition to which the notification applies. In the following subsections, we will further describe the semantics and usage of the Condition option, illustrating the capabilities of this extension.

2.4.1 The Condition Option Format

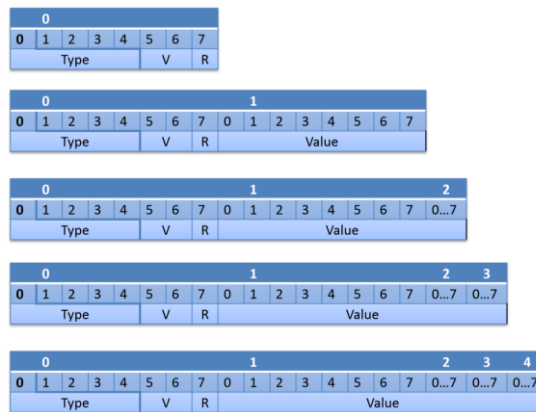


Figure 2-6: Format of the option value of the Condition Option

The condition option is an elective and proxy unsafe option ([2.4], [2.15]). The option value (see Figure 2-6) may have length between 1 and 5 bytes. The most significant 5 bits of the first byte indicate the condition type allowing up to 32 different condition types; the following bit is reliability flag indicating if the response should be acknowledged or not and the last two bits indicate the type of the value in the following bytes. Currently, integer, float and duration are identified as condition value types. The subsequent bytes, which are optional, store the conditional values to be exchanged.

2.4.2 Condition Types

[2.15] identifies 9 condition types, some of which are Time-based while others are value-based. Minimum Response Time, Maximum Response Time, and Periodic option types are time-based conditions whereas AllValues<, AllValues>, Value=, Value<> and Step use the sensor reading values as notification criteria. The Time Series condition type is neither related to time nor to sensor readings.

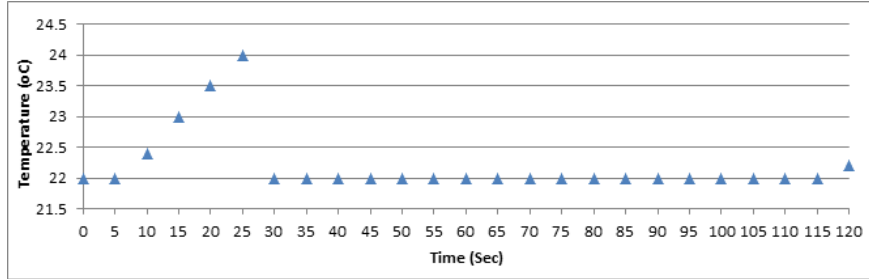


Figure 2-7: Temperature (°C) Data over 120 Seconds

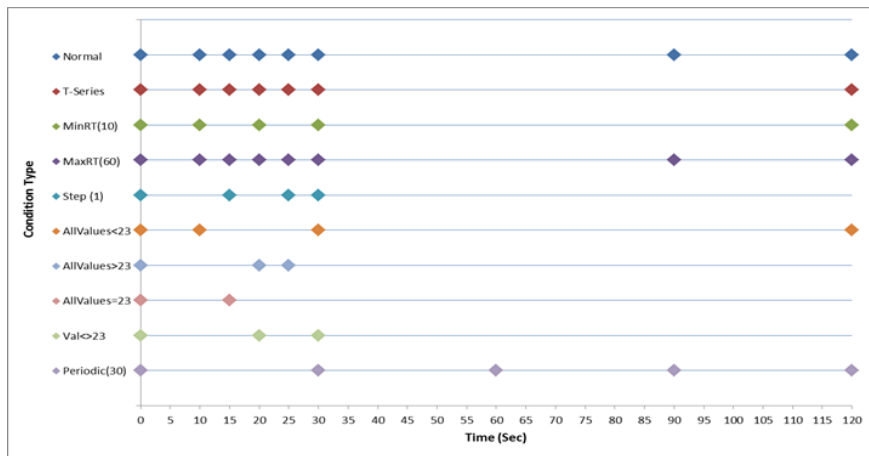


Figure 2-8: Notifications Generated While Using Different Condition Types

To further illustrate how different condition types generate notifications we show an example where a client and a server node establish a temperature observation relationship. Sensor readings drawn every 5 seconds will be notified to the client depending on various conditions. Figure 2-7 shows the temperature (in °C) and the time the data is drawn from the sensors. In the figure, the triangles indicate the sensor reading values. For instance, the graph shows that when the first GET request was sent (at Time 0), the temperature was 22 and after 5 seconds the value is still the same. The next figure (Figure 2-8) represents which notifications are generated for different condition types using small diamonds. For the purpose of this illustration, the CoAP Max-Age option value is set to the default value of 60 seconds. This means that, for normal observe, the client must be notified if the last notification was 60 or more seconds ago irrespective of the resource state change, as described in [2.14]. For the sake of comparison, we will first present the notification trend when using normal observe.

2.4.2.a Normal Observe

According to the CoAP draft document [2.4], a server sends notifications to observers in three cases. First, a notification is sent to clients when the observation relationship is established for the first time to indicate that the client is added to the observers list. Second, whenever the resource state changes the server sends notifications. Finally, a notification is also sent when the data previously sent to the client is not fresh as indicated by the CoAP Max-Age option which by default is set to 60. In such cases, the server sends the notification, if the previous notification is older than the Max-Age value (even if the resource state stays the same).

Accordingly, given the values in Figure 2-7, the server sends notifications at the establishment of the observation relationship (at time 0); every time when the value changes (at times 10s, 15s, 20s, 25s, 30s, and 120s), and when the Max-Age expires (at time 90s) as shown in the top row of Figure 2-8.

2.4.2.b Condition Type 1: Time Series

With Time Series condition type, every change of resource state triggers notification. The notification criteria are similar to normal observation. The only difference is that time series option ignores the CoAP Max-Age option while normal observe sends a notification when the Max-Age timer expires.

The T-Series row of Figure 2-8 shows the packet transmission for conditional observation type Time Series. According to the sensor readings of Figure 2-7, the client is notified at time 0 (during establishment of the relationship) and at times 10, 15, 20, 25, 30, 120 (when the resource state changes).

2.4.2.c Condition Type 2: Minimum Response Time (MinRT)

When the condition type Minimum Response Time (MinRT) is used in observation relationships, the server sends notification by leaving a fixed minimum amount of time between successive notifications. This condition type is highly valuable for systems where the value changes up and down very frequently and the observer is not interested in every change. Consequently, the server does not always send notifications every time the resource state changes.

The MinRT(10) row of Figure 2-8 shows a relationship where the client requests the server to be notified about state changes, but leaving at least 10 seconds between notifications. In this case, the client sends notifications at time 0 (during establishment of relationship), at time 10s, 20s, 30s and 120s. If we closely look at the values at times 15s and 25s, the values are changed after previous notifications but since the difference between the current time and the last notification time is less than 10 seconds, there will be no notification sent to the clients at those times. Also note that, the Max-Age option has no impact here.

2.4.2.d Condition Type 3: Maximum Response Time (MaxRT)

For this condition, the value specified in the condition value field gives the maximum time in seconds the server is allowed to leave between subsequent notifications. What this means is that the server has to send notifications in 3 cases. First, just like all other condition types and normal observation, at the beginning of the observation relationship; second, whenever there is a resource state change; and third when there is no state change but the maximum response time is reached.

The MaxRT row of Figure 2-8 shows the notification pattern for a client requesting notification by setting Maximum Response Time to 60 seconds. Accordingly, the server notifies the client at time 0 (initial notification), at time 10s, 15s, 20s, 25s, 30s, and 120s (notification due to value changes), and at time 90 (notification due to maximum response time). This condition type, in a way, is similar to normal observe with Max-age set to 60.

2.4.2.e Condition Type 4: Step

Depending on the environment where the server node is deployed, the state of a resource might change so frequently that excessive packets are generated. However, the changes may not be significant enough for the client to trigger any action. In such cases, the client may inform the server to send notifications only when the change is more than a specific value by using the Step condition type. In the Step(1) row of Figure 2-8, the client informs the server to send notifications only when the change in value is greater than or equal to 1. As a result, notifications are only sent at time 0, 15s, 25s and 30s. Since the other changes are not significant enough, the server does not send notifications.

2.4.2.f Condition Type 5: AllValues<

In many cases, clients are not interested in state changes which result in values above a specific threshold. For example, to turn on a heater, the temperature should be below a specific threshold. In such cases, the sensor node responsible to regulate the behavior of the heater is not interested in values which are above the threshold. Hence, they may indicate this preference by using the AllValues< (All values less).

In the AllValues< row of Figure 2-8, we can see that the client is interested to get notified only when the resource state changes result in value below 23. Thus, notifications are sent at time 0, 10s, 30s and 120s.

2.4.2.g Condition Type 6: AllValues>

This condition type is similar to condition 5 above. The only difference is that the notification is sent only when the new value exceeds a threshold set by the client. As Figure 2-8 illustrates, the client is interested to receive notifications only when

the resource state is changed and the resulting value is above 23. Consequently, the server sends notifications at time 0, 20s and 25s only.

2.4.2.h Condition Type 7: Value=

This condition indicates that a client is only interested in receiving notifications whenever the state of the resource changes and the new value is equal to the value specified in the condition value field. In our example of Figure 2-8, the client is interested in values equal to 23. This means, the server has to send notifications only when the value changes and the new value is 23. Therefore, the notifications are sent at time 0, and 15 only.

2.4.2.i Condition Type 8: Value<>

Some applications might require the values they are monitoring to be constant. In health care system, machines at Intensive Care Units (ICUs) the machines that monitor a patient's vital signs could be a good example. In such cases, there are vital signs including, body temperature, heart beat, and blood pressure, that must be constant showing that the patient is in a good condition. However, if the values differ from the specified value, it might indicate the patient needs attention. The Value<> (value different from) condition type indicates that the client should be notified when the value changes and is below or above the specified threshold. Once the notification has been sent, no new notifications are sent for subsequent state changes where the value remains higher or lower. As such, a single notification is sent whenever a threshold is passed in either direction.

The Value<> (23) row of Figure 2-8 shows that the client needs to be notified only when resource state changes result in values other than 23. As a result, notifications are sent at time 0, 20s, and 30s. Notification at time 0 is the initial transmission, notification at time 20 is sent because that was the first change that deviates from 23 (and it was below, 23), notification at time 30 was sent because it was the first time the value goes below 23. The value changes at 10 seconds was not sent because it is still below the threshold and value at 25 was not notified because it is still above the threshold (which was notified at time 20)

2.4.2.j Condition Type 9: Periodic

Many environment monitoring applications may require receiving notifications periodically despite the resource state change. Such applications may use the Periodic condition type along with the period of notification. The Periodic (30) row of Figure 2-8 shows notification trends where a client requires to be notified every 30 seconds. In this example, notifications are sent at time 0, 30, 60, 90, and 120.

One can see clearly that, depending on the condition of interest, a different number of notifications will be transmitted over the constrained network. The exact number will depend on the condition type and, if present, the value in the condition option.

2.5 Implementation

Our implementation of conditional observation is based on **Erbium (Er)** – a low-power REST Engine for Contiki developed by Matthias Kovatsch together with *Swedish Institute of Computer Science (SICS)*. The Erbium REST Engine includes a CoAP implementation that supports CoAP drafts 03, 12 and 13. It also supports block-wise transfers and resource observation [2.16]. To support normal observation, Erbium employs two different mechanisms at the server side. The first mechanism uses timers that are used to periodically check states of resources and notify observers whenever there is a state change. The other mechanism is event based. Whenever an event (e.g. change in temperature) occurs, an event handler will be called, which, in turn, calls a function that notifies registered observers. In the remainder of this paper, we have used timer based, periodic checks for resource changes.

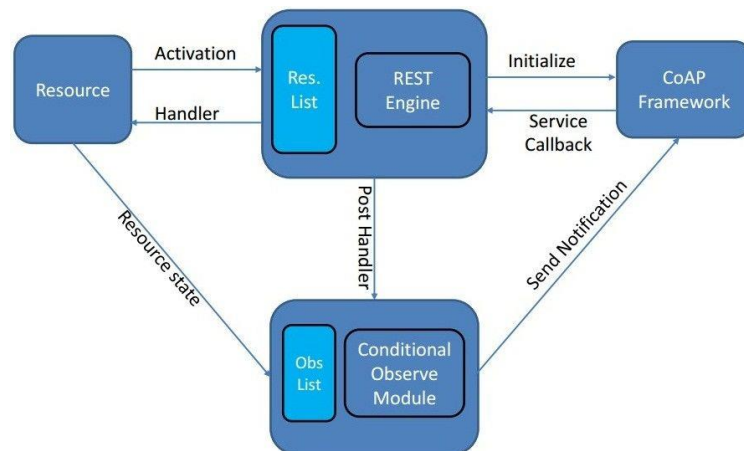


Figure 2-9: Architecture of Erbium

We extended this CoAP implementation to support the new Condition Option and provided some resources that allow conditional observations. Figure 2-9 is a high level architectural diagram of Erbium running on a server node handling normal or conditional observation. The architecture consists of several components, namely, the resources, the REST Engine, the CoAP (and/or HTTP framework), and optional modules such as (Conditional) Observe Module. The REST Engine

is responsible for initializing the CoAP Framework, to store a list of activated resources and to communicate with the optional modules. A conditional observe request received by the CoAP framework will be handled by a service callback function which is declared in the REST engine. The REST engine uses the corresponding handler function to access the states of the resources. As the request is an observation request, the client needs to be registered as an observer in the Conditional Observation Module for future notifications by calling a Post Handler Function. The generation of the first response and subsequent notifications are handled by the CoAP framework. For subsequent notifications, the registration of a single observer will trigger the activation of a function that periodically checks for resource state changes and informs all registered observers. The period is defined for each observable resource separately upon initialization of the resource.

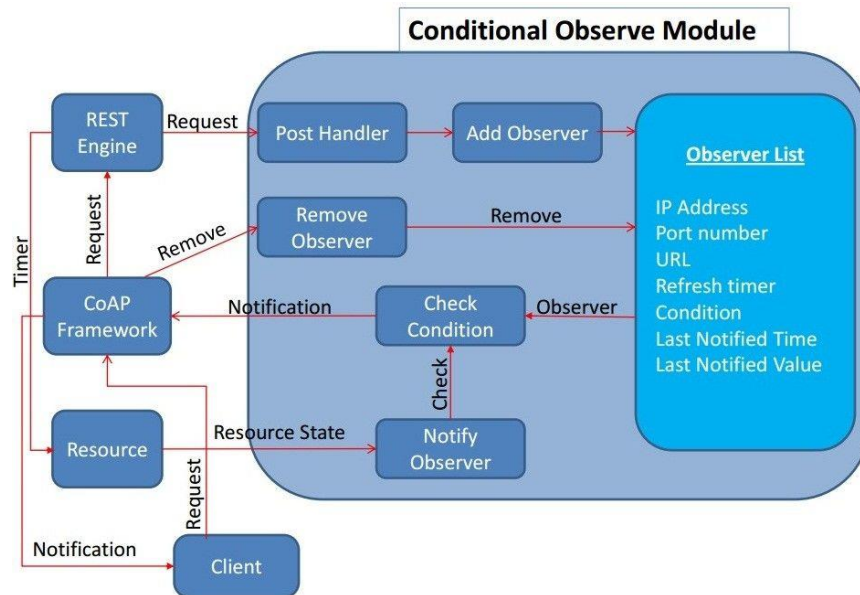


Figure 2-10: Conditional Observation Module

Figure 2-10 is a zoomed-in architectural diagram of the Conditional observation module. When a client sends a conditional observation request, the CoAP framework will receive it. The framework, after confirming that it is a GET request, will call a callback function in the REST engine. Upon receipt of the request, the REST engine does two things. First it prepares the first response by using the predefined handler function and calls a post handler function to add the observer to the observer list. For each observer, the IP address, port number, URI, and refresh timer are stored for normal observe, while for conditional observation also condition information, last notified value and last notification time are stored.

The REST engine then periodically checks for resource states and calls the Notify Observer function (which is part of the conditional observe module) to check if the new value satisfies the filtering criteria set by the client. If it does, the CoAP framework sends the notification to the respective observer(s). Similarly, if a client wishes to stop an observation relationship it sends a normal GET request to the specific resource which will be received by the CoAP framework and will be sent to the Conditional observe module to be removed from the list.

One of the constraints of smart objects is memory. One may wonder about the changes we needed to make to the existing implementation and the code overhead introduced to achieve this additional functionality. As mentioned above, the original Erbium Implementation supports Normal Observe [draft-08]. Our implementation requires additional RAM to store additional observers' information such as observation condition (condition type, value type, reliability flag and condition value), last notification time and last notified value. In addition, it requires more ROM to store instructions that are used to check if the new resource state satisfies the specified condition. Table 2-1 shows the TEXT, Data and BSS section requirements of both Normal Observe (the original implementation) and conditional observe. Note that the conditional observe implementation also encompasses normal observe functionality.

Table 2-1: Memory requirements of Normal and conditional observe

	Text (Byte)	Data (Byte)	BSS (Byte)	Total (Byte)
Normal Observe	50398	386	6050	56834
Conditional Observe	51096	386	6072	57554
Delta	698	0	22	720

It can be seen from the table that the Text segment (ROM) requirement for Conditional Observe is slightly larger than Normal Observe. Similarly, the size of the BSS segment, which stores uninitialized variables, is larger in case of conditional observe just by a few bytes. As our findings in the following sections illustrate, 720 bytes overhead is affordable for the advantage that can be gained through the use of conditional observation, either from a performance viewpoint or from an application developer viewpoint.

2.6 Evaluation

2.6.1 Scenario 1: Basic Evaluation

We used different scenarios to illustrate the relevance of conditional observe as an extension to the normal observe functionality of CoAP. In the first set of experiments, we used Zolertia Z1 motes to be used as client and server nodes in Cooja. To capture the impact of network size on performance, we used between 0 – 6 intermediate Z1 nodes, which merely exist to act as routers between the client and server nodes. We selected the AllValues> (value based) and Periodic (Time based) condition types to compare the performance against Normal Observe. For every hop, and every condition value we run the test 10 times to average the results. For sensor values, we generated 288 pseudo-random numbers between 17 and 26 (representing temperature values). The average of the values is 20. For AllValues> condition type, we tested three condition values: the minimum (17), the average (20) and the maximum (26). Every 5 seconds, the server is made to retrieve a value from an array of 288 numbers sequentially as a new sensor reading. As CoAP supports both confirmable and non-confirmable requests, we repeated the same experiment twice to see the impact of reliable communication on network performance.

Figure 2-11, shows the number of packets transmitted for different condition types while Figure 2-12 shows the power consumption where the requests are sent as non-confirmable. Figure 2-13 shows the power consumption in the case of confirmable communication, which is slightly higher than the case of non-confirmable notifications due to the additional acknowledgements and potential retransmissions in case of packet loss.

We may learn two basic lessons from Figure 2-11, Figure 2-12 and Figure 2-13. First, this simple scenario shows that the solution we proposed works and is implementable in constrained devices such as Z1 motes. This is demonstrated for both value-based and time-based conditions, which require a different implementation approach. Second, for such a simple scenario, using normal observe as mechanism to collect all resource state change in combination with client side filtering generates a larger amount of packets as compared to all other conditional observation methods. This leads to higher power consumption and, hence, low battery life. From these findings we can conclude that, even though the exact impact is heavily dependent on specific use cases, conditional observation can be considered a useful extension to normal observation.

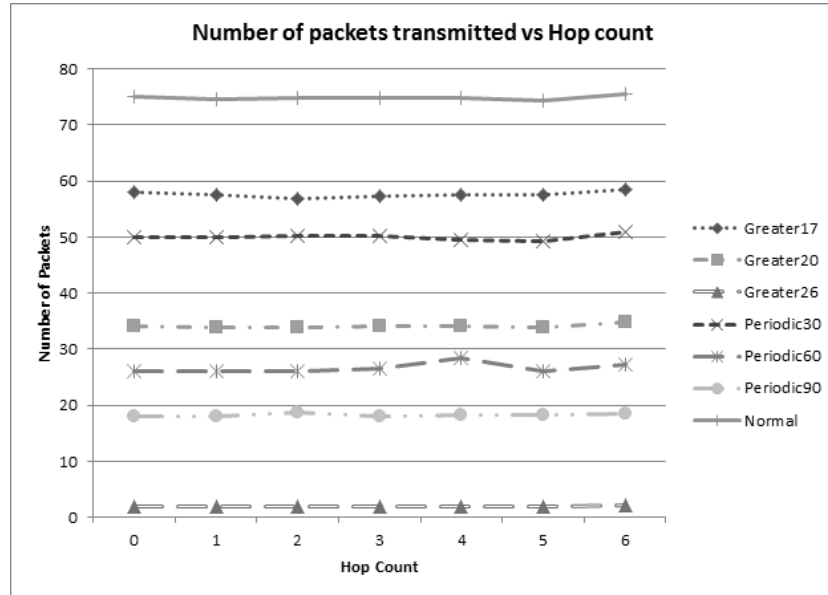


Figure 2-11: Number of Packets transmitted Vs. Hop count

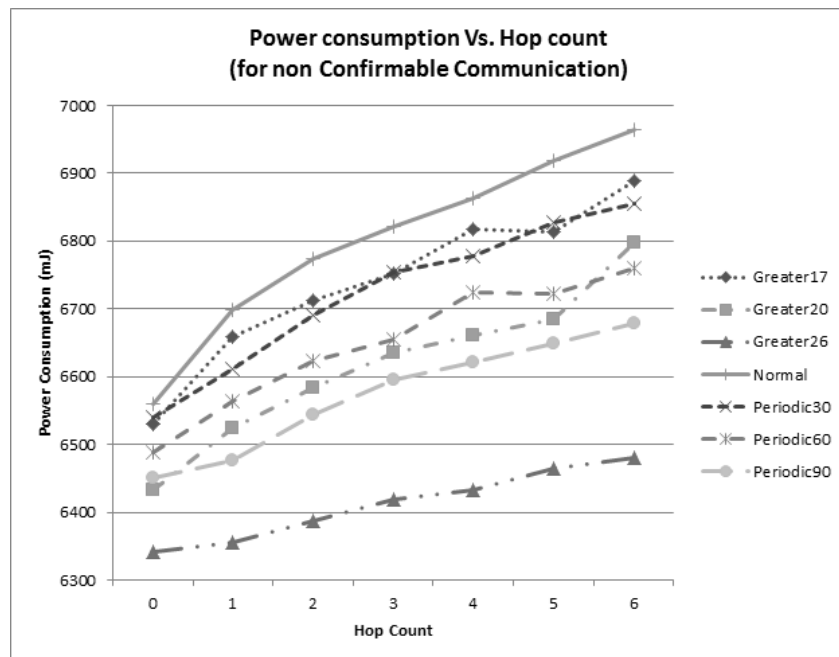


Figure 2-12: Power Consumption Vs. Hop Count (Non Confirmable Transmission)

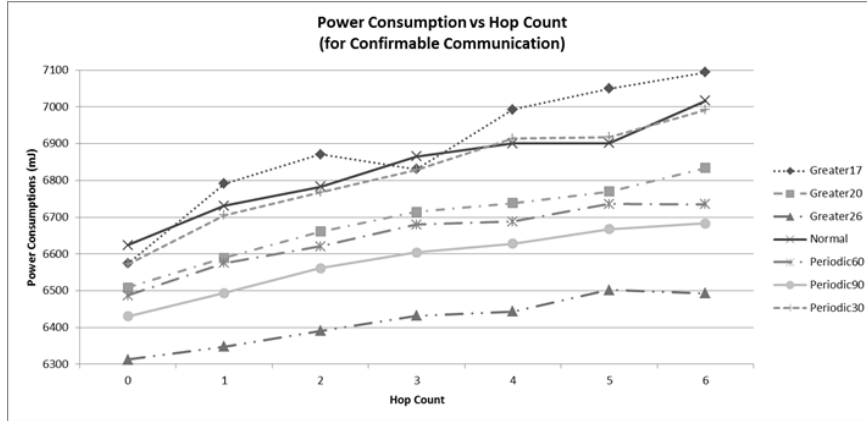


Figure 2-13: Power Consumption of nodes (Confirmable Transmission)

2.6.2 Scenario 2: Non-constrained Client - Gateway – Multiple servers

In most applications, the clients run on non-constrained devices such as normal computers, smart phones or similar other devices. To illustrate the use of conditional observation in scenarios where multiple constrained servers are communicated from a non-constrained device and network, we combined Cooja servers running Erbium with our COAP++ client, part of our modular C++ CoAP framework developed in Click Router [2.17]. We used 2 servers and a border router running Contiki in Cooja. We established a tunnel between the border router and the computer so that the COAP++ client can communicate with the Cooja servers.

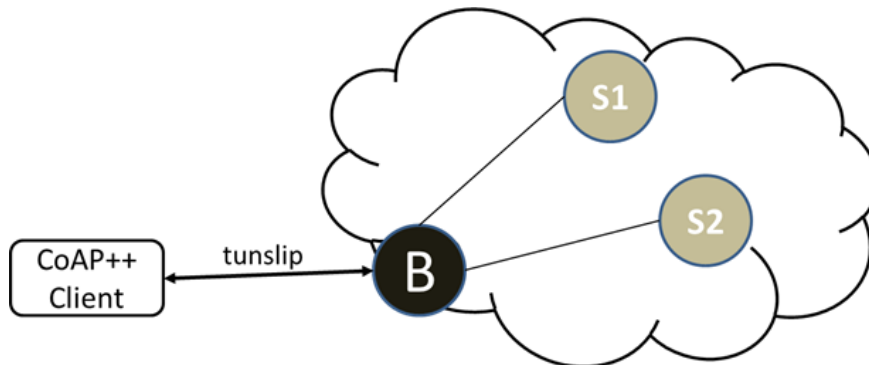


Figure 2-14: Experimental setup consisting of non-constrained client, and 2 constrained servers

Figure 2-14 shows the connection between the sensor nodes and click. As in the case of the above test, Z1 nodes were used in Cooja and AllValues> condition

type was used to compare the result with normal observe. The data to be generated by the servers is the same pseudo-random numbers used in the previous tests. The test was run 10 times with condition value 20, which is the average of the data set. The test showed that the average power consumption of the two servers is **4787mJ** and **4936mW** for normal observation and **4672mJ** and **4856mJ** for conditional observation. These results confirm the results we achieved in the previous scenario.

The tests above show that the new implementation can be run on constrained devices such as Zolertia Z1 motes which have **8KB RAM** and **92KB ROM**. The slight increase in memory requirement, especially to filter packets on the server, is an affordable trade-off to gain substantial energy saving and avoid congestion of the constrained networks.

2.6.3 Mathematical Evaluation

The previous tests show that conditional observation can achieve a significant gain in terms of the reduction of the number of transmitted packet and subsequently of the reduction in power consumption of constrained devices. Of course, the exact gain depends largely, amongst others, on the frequency of resource states and the specific conditions interested in. In this subsection, we will show this potential gain of conditional observation through a mathematical evaluation.

The total power consumed in a given period by a device, E , is the sum of the consumption of the radio and the microcontroller chips. For simplicity, we will assume that the power consumption of other peripheral devices, such as sensors, is insignificant. Therefore,

$$E = E_{Radio} + E_{Processing}$$

The Radio could be either in active transmitting (Tx), active receiving (Rx), idle transmitting or idle listening state and the microcontroller will be either in Active Mode (CPU-Active) or Low Power Mode (CPU-LPM). Assuming that the radio consumes equal amount of power during idle listening and idle transmitting state, we have:

$$E = E_{Tx} + E_{Rx} + E_{IDLE} + E_{CPU-Active} + E_{CPU-LPM}$$

Energy, power consumption in a given period of time, can be computed as,

$$Energy = Power \times Time$$

Therefore, the total power consumption in a particular period of time will be

$$E = P_{Tx} \times T_{Tx} + P_{Rx} \times T_{Rx} + P_{IDLE} \times T_{IDLE} + P_{CPU-Active} \times T_{CPU-Active} + P_{CPU-LPM} \times T_{CPU-LPM}$$

and

$$T_{Tx} + T_{Rx} + T_{Idle} = T_{CPU-Active} + T_{CPU-LPM}$$

The power consumption values are all known from the datasheets of the chips of the constrained devices, the time values for the radio can be derived from the MAC protocol and the number of packet transmissions and receptions and the time values for the CPU can be derived from experimental results. To compare the power consumption difference between normal and conditional observe and to keep the mathematical model sufficiently simple, we make the following assumptions:

- We use value-based condition types
- The resource state changes every S seconds
- The device uses a duty-cycled Low Power Listening protocol. The length of the duty cycle is L and the duty cycling value is d (only $d\%$ of the time the radio is active, listening for incoming packets in order to save energy). When the device needs to transmit a packet, it has to turn on the radio for the entire period L .
- The probability that the condition is not fulfilled is p
- The value of Max-Age is equal to 60 seconds.
- The transmission of a notification requires only a single packet
- The notifications are non-confirmable messages, so the server generating the notifications is not receiving any packets.

For Normal observe, every S seconds the device will do some processing and transmit a notification. In case S becomes larger than Max-Age, a notification is also sent every time Max-Age expires. This brings the total number of notifications sent, N , equal to $\text{CEIL}(S/\text{Max-Age})$.

Hence, for normal observe, $T_{CPU-LPM}$ is

$$T_{CPU-LPM} = S - T_{CPU-Active}$$

For Conditional Observation the transmission of a packet in the interval S depends on the condition value and is probabilistic. We know that conditional observations require some additional processing for checking the condition. However, in case no notification must be sent, no processing is needed to prepare the packet transmission. To incorporate both effects, we introduce Δ_{proc} giving an estimate for the difference in processing. Thus, the time, $T'_{CPU-LPM}$ is

$$\begin{aligned} T'_{CPU-LPM} &= T_{CPU-LPM} - \Delta_{proc} = S - T_{CPU-Active} \\ &= S - (T_{CPU-Active} + \Delta_{proc}) \end{aligned}$$

Similarly, the time the radio chip spent in the different states, can be computed from the period S , the duty cycle length L the duty cycling value d and the number

of packets to be transmitted. For Normal Observation, the time for TX, RX and Idle states is given by:

$$\begin{aligned} T_{Rx} &= (S - N \times L) \times d \\ T_{IDLE} &= (S - N \times L) \times (1 - d) \\ T_{Tx} &= N \times L = S - (T_{IDLE} + T_{Rx}) \end{aligned}$$

Here, we have assumed that if a device has to transmit packets, it will use the whole period L for the transmission. The formula for conditional observation will have to take the probability of transmission into consideration. For probability value p (condition not fulfilled), the time spent will be:

$$\begin{aligned} T_{Tx} &= 0 \\ T_{IDLE} &= S \times (1 - d) \\ T_{Rx} &= S \times d \end{aligned}$$

And for probability values 1 – p (condition fulfilled), the overhead will be the same as normal observation, with N equal to 1. Therefore, the total energy consumption, based on our equations above, during S seconds for Normal Observe, O(S) and Conditional Observe, CO(S) will be:

$$\begin{aligned} E &= P_{Tx} \times T_{Tx} + P_{Rx} \times T_{Rx} + P_{IDLE} \times T_{IDLE} + P_{CPU-Active} \times T_{CPU-Active} \\ &\quad + P_{CPU-LPM} \times T_{CPU-LPM} \\ O(S) &= P_{CPU-Active} \times T_{CPU-Active} + P_{CPU-LPM} \times (S - T_{CPU-Active}) + P_{Tx} \times N \\ &\quad \times L + P_{Rx} \times (S - N \times L) \times d + P_{IDLE} \times (S - N \times L) \times (1 - d) \\ CO(S) &= P_{CPU-Active} \times (T_{CPU-Active} + \Delta_{Proc}) + P_{CPU-LPM} \\ &\quad \times (S - T_{CPU-Active} - \Delta_{Proc}) \\ &\quad + p \times [P_{Tx} \times 0 + P_{Rx} \times S \times d + P_{IDLE} \times S \times (1 - d)] \\ &\quad + (1 - p) \times [P_{Tx} \times L + P_{Rx} \times (S - L) \times d + P_{IDLE} \times (S - L) \times (1 - d)] \end{aligned}$$

In order to be able to evaluate the above values for different parameter values of S and p, we used the parameters shown in Table 2-2.

Table 2-2: Parameter Values

Parameter	Value	Explanation
$I_{CPU-LPM}$	0.0005mA	Taken from the data sheets of the Z1 mote.
$I_{CPU-Active}$	8mA	
I_{Rx}	18.8mA	

Parameter	Value	Explanation
I_{Tx}	17.4mA	
I_{IDLE}	0.426mA	
V	3V	
$P_{CPU-LPM}$	0.0015mW	
$P_{CPU-Active}$	24mW	Power = current * voltage
P_{Rx}	56.4mW	
P_{Tx}	52.2mW	
P_{IDLE}	1.278mW	
L	125ms	Based on ContikiMAC LPL
D	0.01	
Δ_{proc}	1ms	Approximation based on extensive experiments.
CPU active	1.48%	Average percentage of time the CPU is active, based on extensive experiments.

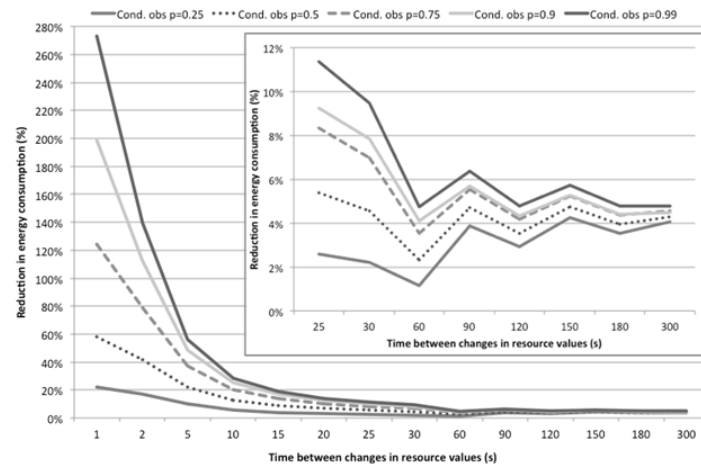


Figure 2-15: Power Consumption Vs. Probability of Packet Transmission

From the values $O(S)$ and $CO(S)$, one can easily calculate the energy consumed during 1 second and compare the difference in energy consumption between

normal observe (assuming the collection of all values using normal observe and client side filtering) and conditional observe. Figure 2-15 shows the reduction in energy consumption that can be achieved this way, for different values of S and p .

We see that for increasing values of the probability p , i.e. the probability that the condition is not fulfilled, the reduction in energy consumption also increases. For a fixed time S between resource changes and thus a fixed amount of potential notifications, an increasing probability p implies that less notifications have to be sent compared to normal observe, leading to a reduction of the energy spent on transmitting these notifications and thus of the overall energy consumption. This is in line with our experimental evaluation. Further, we notice that, for a fixed value of p , smaller values of S , the time between resource changes, leads to major reductions in energy consumption. On the other hand, for larger values of S , the potential energy reduction gradually decreases as can be seen from the zoomed in area of the chart. For values larger than 30 seconds the average reduction varies between approximately 2% and 6%. The fact that the curves periodically rise and fall is due to the impact of the MAX-AGE value, which causes the number of notifications to be sent by normal observe to be a step function expressed by $\text{CEIL}(S/\text{MAX-AGE})$.

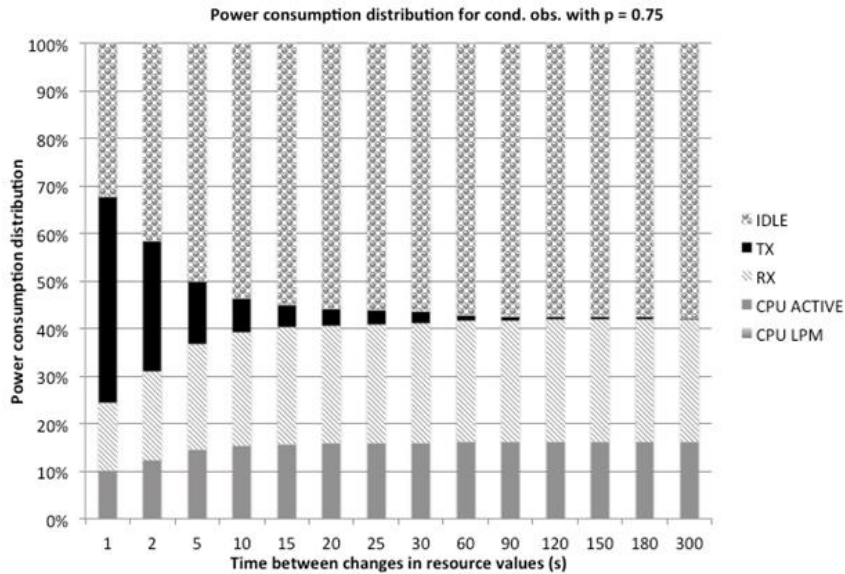


Figure 2-16: Distribution of the power consumption over all different energy consumers

The impact of S on the energy reduction can be further explained by looking at the contribution of all different energy consumers to the overall energy consumption as shown in Figure 2-16. For frequently changing resource values and thus more

frequent notifications for a given value of p , the power consumed for transmitting notifications makes up a large part of the total energy budget. For increasing values of S , and thus less notifications, this part becomes smaller and smaller compared to other energy consumers such as the idle energy consumption of the radio. Since conditional observe almost solely impacts the TX part of the energy budget, its impact is reduced for larger values of S . Of course, it should be noted that the frequency with which values (e.g. sensor readings) can change also depends on the granularity of the measurements and/or the application on the device. For instance, the device can be programmed to retrieve the latest sensor reading only every 5 minutes, but it could also read out the temperature every 10 seconds. Further, if the granularity of the readings is higher, e.g. a granularity of 0.1 Centigrade instead of 1 Centigrade, readings will more often result in notifications.

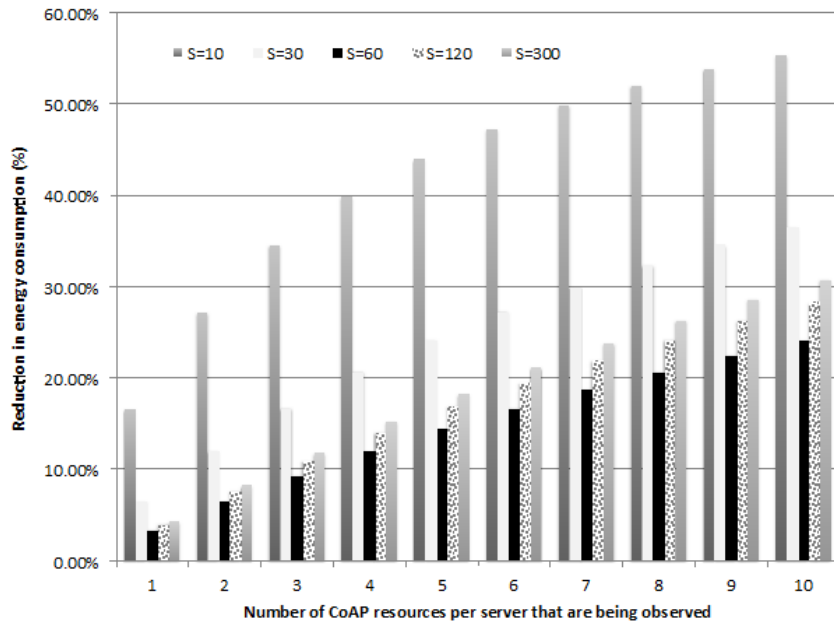


Figure 2-17: Reduction in energy consumption of using conditional observations with $p=0.75$ versus normal observe for a varying number of resources on the server

The above mathematical evaluation reveals that a mechanism such as conditional observe is extremely useful for resources that change very frequently. One could (falsely) conclude that it is not that useful for larger values of S . However, this is not true. First of all, the concept of conditional observations remains very useful for application developers, which are now offered easy to use primitives to collect sensor data based on conditions, and can serve as an enabler for IoT applications. Next to this, the above mathematical evaluation has been made in the assumption

of a single server with a single resource that sends its notifications directly to the sink or gateway of the sensor network. In case a single constrained device hosts multiple resources (temperature, light, humidity...) that are conditionally observed, one will experience the combined effect of having fewer packet transmissions for every resource individually. This means that even for larger values of S a significant energy reduction can be achieved. This effect is illustrated in Figure 2-17. When the number of resources per device is increased, all resources being observed conditionally and having their state changed every S seconds, the reduction in energy consumption remains significant even for higher values of S . It also worth noticing that the gain for S equal to 60 is smaller than the gain for larger values of S . This is because of the impact of MAX-AGE on the number of notifications sent using normal observe.

Further, in case one of the resources has a smaller MAX-AGE value than the default value of 60 seconds, the reduction also becomes bigger: between 2% and 6% for S values between 30 and 300 seconds and MAX-AGE equal to 60 seconds, between 4 and 10% for S values between 30 and 300 seconds and MAX-AGE equal to 30 seconds and between 20 and 30% for S values between 30 and 300 seconds and MAX-AGE equal to 10 seconds. Last but not least, there is also the effect of larger-scale, multi-hop networks with multiple servers. In this case, every single notification will result in multiple packet transmission through the network, increasing the TX energy consumption in all intermediate nodes and thus contributing to the overall reduction of the network lifetime.

A last aspect that has not been discussed so far is the impact of multiple conditional observation relationships on a single resource (i.e. by different clients). For normal observe and in the presence of an intermediary, the intermediary can aggregate multiple observe relationship into a single one. This means that the intermediary establishes an observe relationship itself on behalf of multiple clients and delivers the resulting notifications to all clients. This way, notifications have to travel through the network only once. When using conditional observations in the presence of an intermediary, the possibility to aggregate different conditional observations into a single conditional observation relationship strongly depends on the condition type and associated values. In this case it is possible that no optimal aggregation can be found, reducing somewhat the overall performance.

2.7 Use Cases

From the previous discussion, it has become clear that the actual advantage in terms of performance gain (energy reduction) of conditional observations is heavily dependent on specific use cases. Apart from that, there is the additional

advantage that it offers easy to use primitives to collect sensor data of interest, which have a small footprint and which are reusable by all CoAP resources hosted on a device. To concretize the advantage of having conditional observation as an extension to CoAP, we will now discuss in more detail three real-life IoT use cases.

2.7.1 Heating and Cooling Systems

Smart buildings, heavy machineries and greenhouses use temperature sensors to regulate their environment. Input from sensors will be used by heating and cooling systems, i.e., the actuators, to take the necessary actions to maintain the temperature at a specified level. If the temperature exceeds beyond a certain level, cooling systems at particular locations need to be activated to lower the temperature. Similarly, if it is below a certain threshold, a heating system has to increase the temperature.

When using wireless embedded systems to monitor and control the environment, conditional observation may play a significant role in such systems. Consider a building equipped with such a system. If the temperature in the building has to be maintained between 19°C and 22°C, the sensors have to inform either the heater or the cooler if the temperature is out of this range. To realize this, the heating or cooling system that is linked to a sensor may send two conditional observation requests. The first condition type should be `AllValues<19` and the second should be `AllValues>22`. When the server receives these requests, on separate port numbers, it stores them as two different requests and sends notifications to the client when one of the two conditions is fulfilled.

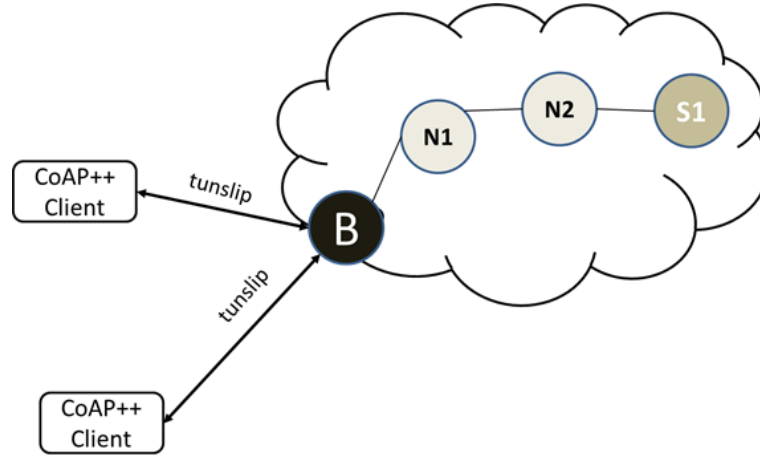


Figure 2-18: Experiment setup with 2 Click++ clients, a Contiki border router, a Contiki Server and 2 Intermediate nodes.

To test this scenario using our implementation, we performed a test involving two instances of CoAP++ clients, a Cooja Border Router, Cooja CoAP server, and 2 intermediate nodes as shown in Figure 2-18. We used real life temperature data collected at Intel Laboratories. To simplify the test, we used a one-day data collected every 5 minutes starting from midnight. The 288 data points were sent by the CoAP server every 5 seconds when the simulation starts. We repeated the experiment 10 times to average the result. The final result shows that, Normal Observe consumed 4936mJ while Conditional Observe consumed 4716mJ showing an improvement in power consumption by using conditional observation. The above example can be easily extended to other smart building applications involving a variety of sensors that need to be observed.

2.7.2 Smart Environment Monitoring

There is a growing concern of pollution everywhere in the world. Pollution, be it air pollution, water pollution or land pollution, is the introduction of harmful substances to clean sources (air, water, etc.). Air Quality Index (AQI) is widely used to measure the level of pollution of air by different pollutants such as ground level ozone, particulates, sulfur dioxide, carbon dioxide, carbon monoxide and nitrogen monoxide. The AQI values fluctuate substantially depending on various situations. Most countries divide the AQI values in different categories and take different actions depending on the level of AQI.

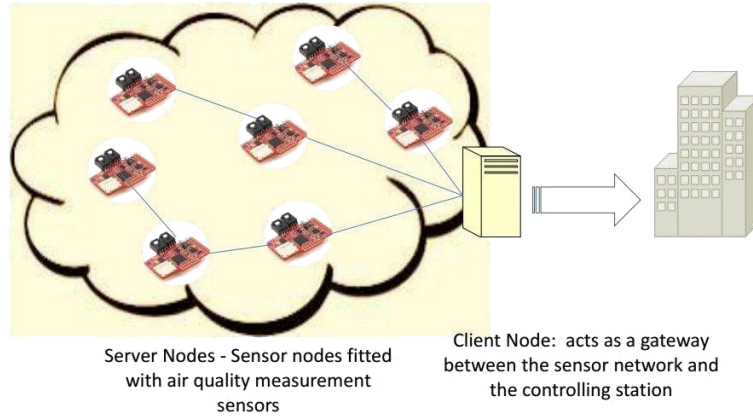


Figure 2-19: Air Quality Controlling Setup

Smart environment monitoring applications make use of Wireless Sensor Networks (WSNs) to proficiently monitor the pollution level and come up with the AQI level of the environment. Such applications can also benefit from conditional observations in order to realize the desired behavior. Consider a simple environment monitoring system aimed at collecting the concentration of pollutants (e.g. CO_2) in a particular area and communicating it to a central station. This solution may be implemented in various ways. One such implementation is depicted in Figure 2-19. Sensor nodes (servers) are connected with each other and to the gateway node through wireless links. The gateway connects the central station with the servers. Every sensor node collects the data and sends it to the gateway node, which, in turn, communicates it to the central station. If no (conditional) observation is employed, the gateway would have to poll values from the sensors every fixed interval or whenever the need arises.

Using conditional observation, much more flexibility is introduced and system efficiency can be improved. The client (e.g. the gateway or an application in the cloud) may establish conditional observation with the servers (sensor nodes) stating its interest to be notified periodically. Here, the *periodic* condition type can be used for the subscription. Once this observation relationship is established, the servers will generate notifications periodically. It is also possible to further refine notifications based on prevailing circumstances. For example, in normal situations, where the concentration of pollutants is very low, there is no need to send notifications to the gateway very frequently. So, the notification interval can be set, for instance, to 1 hour. However, as soon as the pollution level increases, which can be detected by establishing another conditional observation, the client may opt for more frequent updates (say every 5 minutes) by sending an updated

conditional observation request which eventually removes the old observation relationship and establishes a new one with a higher frequency.

2.7.3 Sleepy Nodes

Sleepy nodes are devices which occasionally go to a low power mode by cutting power to unnecessary components to save energy. Some devices cut power only to the radio system while the other components run as usual. At any time, the device could be at a sleeping state or awake. But, in most cases the sleeping time is much larger.

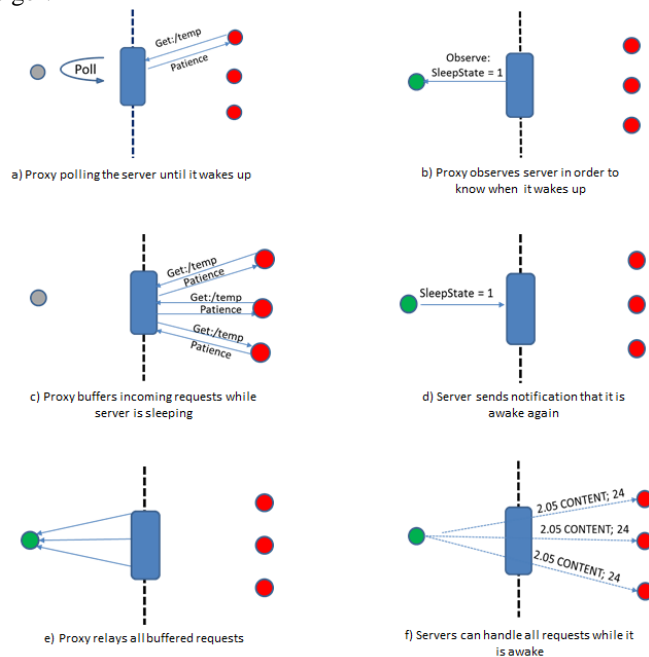


Figure 2-20: Communication with sleepy nodes using conditional observation

Consequently, communication with sleepy nodes is very problematic, especially if the sleepy node has resources that a client needs from time to time. The major reason for this is that, when a node is in the sleep state, it is disconnected from the network and is unreachable. One solution to resolve this issue efficiently is to use proxy nodes and conditional observations. In the proxy model, all clients get connected to the sleepy node through a proxy. The proxy, then, relays client requests to the server. As soon as the server receives a request, it directly sends back a response to the client via the proxy. However, due to the sleepy nature of the server, the communication is not as simple as this, but the conditional observe mechanism offers an elegant solution to this problem as explained in the Figure 2-20. In the figure, the red circles are the clients, the gray circle is the server in

sleep mode (SleepState = 0), the green circle is the server in awake mode (SleepState = 1) and the middle rectangle is the proxy.

- a) A client request arrives at the proxy. The server is sleeping (SleepState = 0). The proxy buffers the request and sends a response to the client telling it to be patient for the actual response. Next the proxy starts to continuously check the server to see if it is awake.
- b) As soon as the proxy detects the server is awake (meanwhile retrieving the value requested by the client and delivering the response to the client), it sends a conditional observe request indicating its need to be notified when the server wakes up (SleepState = 1). This can be achieved by using the VALUE= condition type. The server adds the proxy to the observers list.
- c) After some time, the server may go to sleep for a long time (SleepState = 0). While the server is sleeping, clients may send GET requests to the proxy. Since the proxy now knows the server is sleeping, it buffers all requests and sends back patience responses.
- d) When the server wakes up, it sends a notification to the proxy indicating its sleep state has changed, back to Awake.
- e) When the proxy gets the notification, it sends all buffered requests to the server.
- f) Finally, the server sends the responses directly to the clients. As an optimization the proxy may aggregating similar requests into a single request.

Communication with the sleepy node will be even more efficient if the clients themselves register as observers requesting the server to notify them when a particular criterion is met. In this case, once the server is awake and knows clients' requirement through the proxy, all subsequent notifications will be made directly to the clients whenever the criterion is met. This is done without the involvement of the proxy.

2.8 Conclusion

In this paper, we presented the concept of conditional observations as an extension to the CoAP protocol in general and the Observe option in particular. The design of this concept was driven by the need to have a lightweight, efficient and compact solution for subscribing for very specific events, which is an important functionality when building IoT applications that directly interact with constrained devices. Normal observe in combination with client-side filtering can realize similar functionality, but suffers from the transmission of excessive packets that are not of interest to clients. We demonstrated the feasibility of implementing this

functionality on constrained devices. Using this implementation, we presented comparative results of using normal observation and client-side filtering versus conditional observation. We also presented theoretical evaluations of normal and conditional observation. From both the experimental and theoretical results, it is evident that the conditional observations are very useful extensions to the basic observe behavior, both from an application point of view and from a network efficiency point of view. It enables clients to receive notifications that contain only state changes they are interested in. This has a twofold advantage: an application has the expressiveness to selectively collect data and the data of no interest does not have to travel over the network. The latter advantage will become even more important in larger constrained networks where notifications have to travel over multiple hops. As such, conditional observations can greatly contribute to the reduction of battery consumption and increase of network lifetime. In addition, many scenarios can be thought of that can benefit from this functionality. As such, conditional observation is an interesting and easy-to-use enabler for many IoT applications. As future work we identified studying the impact of multiple conditional observation relationships on a single resource and the design of an efficient solution to aggregate multiple of these relationships into a single one in order to further reduce traffic.

Acknowledgement

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°258885 (SPITFIRE project). We would also like to acknowledge our co-authors of the IETF CoRE conditional observe draft.

References

- [2.1] Z. Shelby, Embedded Web Services, IEEE Wireless Communications, Dec 2010, pp.52-57.
- [2.2] G. Montenegro, N. Kushalnagar, D. Culler, IETF RFC4944 - Transmission of IPv6 Packets Over IEEE 802.15.4 Networks, September 2007.
- [2.3] J. Hui, et. al., IETF RFC6550 - RPL: Routing Protocol for Low Power and Lossy Networks.
- [2.4] Z. Shelby, K. Hartke, and C. Bormann, Constrained Application Protocol (CoAP) draft (draft-ietf-core-coap-13), December 6, 2012 (work in progress)
- [2.5] J. P. Vasseur and A. Dunkels, Connecting Smart Objects with IP: The Next Internet (Morgan Kaufmann 2010).

-
- [2.6] Z. Shelby, RFC6690 - Constrained Restful Environment (CoRE) Link Format.
 - [2.7] A. B. Roach, RFC3265 - Session Initiation Protocol (SIP): Specific Event Notification.
 - [2.8] A. Stanford-Clark, and H. L. Truong. MQTT for Sensor Networks (MQTT-S) Protocol Specification Version 1.1, IBM Corporation, 2008.
 - [2.9] E. Souto, et al., Mires: a publish/subscribe middleware for sensor networks. (Springer-Verlag London Limited, 2005).
 - [2.10] S. Lai, J. Cao and Y. Zheng, PSWare: A publish / subscribe middleware supporting composite event in wireless sensor network.
 - [2.11] European Telecommunications Standards Institute (ETSI). Machine-to-Machine communications (M2M). ETSI TS 102 690 V1.1.1 (2011-10).
 - [2.12] Z. Shelby, and M. Vial, CoRE Interfaces - draft-shelby-core-interfaces-3 (work in progress).
 - [2.13] Open Geospatial Consortium Inc. Sensor Observation Service, 2007.
 - [2.14] Z. Shelby, K. Hartke and C. Bormann. Observing Resources in CoAP - draft-ietf-core-observe-07, 2012 (work in progress).
 - [2.15] S. T. Li, J. Hoebeke, and A. J. Jara, Conditional Observe in CoAP - draft-li-core-conditional-observe-03, 2012 (work in progress).
 - [2.16] M. Kovatsch, S. D. Valencia, A Low-Power CoAP for Contiki: in Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011), (pp. 855-860), 2011.
 - [2.17] I. Ishaq, J. Hoebeke, J. Rossey, E. De Poorter, I. Moerman, P. Demeester, "Facilitating sensor deployment, discovery and resource access using embedded web services", International Workshop on Extending Seamlessly to the Internet of Things (esIoT), July, 2012

3

Bindings and RESTlets: A Novel Set of CoAP-Based Application Enablers to Build IoT Applications

In many IoT applications, sensors and actuators are expected to work together to achieve a common goal. In order to achieve flexibility in associating a sensor to an actuator, most interactions between the sensors and actuators, however, have to go through an intermediary which establishes an observation relationship with the sensor and triggers actuators based on the data obtained from the sensor. Using an intermediary has several drawbacks including latency, congestion at nodes towards the sink and failure of the whole system if the intermediary fails. Bindings are introduced in this chapter as a means of excluding the intermediary and letting the sensor and actuator interact directly in a flexible way. Extending this concept further, this chapter discusses RESTlets. RESTlets are defined as IoT application building blocks that can be interconnected with other IoT application components (and other RESTlets) through bindings to enable distributed IoT development.

Girum Ketema Teklemariam, Floris Van Den Abeele, Ingrid Moerman, Piet Demeester and Jeroen Hoebeke. *Bindings and RESTlets: A Novel Set of CoAP-Based Application Enablers to Build IoT Applications*. Sensors 2016, 16(8), 1217; doi:10.3390/s16081217

Abstract: *Sensors and actuators are becoming important components of Internet of Things (IoT) applications. Today, several approaches exist to facilitate communication of sensors and actuators in IoT applications. Most communications go through often proprietary gateways requiring availability of the gateway for each and every interaction between sensors and actuators.*

Sometimes, the gateway does some processing of the sensor data before triggering actuators. Other approaches put this processing logic further in the cloud. These approaches introduce significant latencies and increased number of packets. In this paper, we introduce a CoAP-based mechanism for direct binding of sensors and actuators. This flexible binding solution is utilized further to build IoT applications through RESTlets. RESTlets are defined to accept inputs and produce outputs after performing some processing tasks. Sensors and actuators could be associated with RESTlets (which can be hosted on any device) through the flexible binding mechanism we introduced. This approach facilitates decentralized IoT application development by placing all or part of the processing logic in Low power and Lossy Networks (LLNs). We run several tests to compare the performance of our solution with existing solutions and found out that our solution reduces communication delay and number of packets in the LLN.

3.1 Introduction

In the information age that we are living in, we have witnessed remarkable advances in electromechanical technologies, miniaturization and wireless communication. It is not uncommon to see tiny and very powerful electromechanical devices that can be integrated in our environment by embedding them in everyday objects around us and are capable of doing things that were unimaginable a few decades ago. Sensor and actuator nodes are best examples of such tiny devices that have become parts of our daily life for quite some time. Sensor nodes are devices that capture events in the physical world and transfer them into the virtual world as raw data so that they can be processed and acted upon. Actuators, on the other hand, alter the real world based on data obtained from the physical world or cyber world, periodically or spontaneously. It is very common to see sensors and actuators working together. Motion activated doors, air conditioners, environmental monitoring systems, and voice activated security systems are some examples which involve both sensors and actuators. The communication arena has also seen remarkable changes in the past couple of decades. Billions of devices are being interconnected with each other around the globe. These advancements of both communication and electromechanical technologies have led to new possibilities for those tiny devices. Sensors can be connected to other actors such as services that process the raw data and subsequently interact with actuators without regard for physical proximity. In the

beginning, these solutions were characterized by proprietary solutions implemented by different vendors. Such solutions either involve proprietary intermediary devices between the sensors and actuators for data processing or require static configuration to be done to allow direct interactions. The former approach forces users to stick to one vendor due to lack of interoperability. In addition, when using multiple vendors, it results in several vertical silos for different applications. The latter approach also has its own limitation such as limited reuse, complexity in programming, and limited integration with applications. Since the last decade, diverse initiatives have been launched by various organizations in order to make these devices an integral part of the Internet. The Internet Engineering Task Force (IETF) has been the pioneer by establishing a number of focused working groups that address various aspects of the integration. Due to the constraints of the tiny devices such as limited processing, storage, transmission and reception capacity and limited power availability, existing Internet networking protocols were not directly suitable. Therefore, various solutions have been proposed at the different layers of the Open Systems Interconnection (OSI) model. For instance, IPv6 packets are too big to be processed, transmitted, and received in an energy efficient way by the constrained devices. To this end, the 6LoWPAN adaptation layer has been developed for compression/decompression and transmission of IPv6 packets so that they can be transported over constrained networks [3.1]. Another issue that needs to be addressed is routing. The routing protocols used in today's Internet are all designed to find optimal paths to destinations by considering the routers to be powerful enough to store and process large routing tables. This makes them not suitable for constrained multi-hop networks. Moreover, the metrics that should be considered in constrained networks are considerably different from those in non-constrained networks. As a result, a new routing protocol named Routing Protocol for LLN (RPL) has been proposed [3.2]. Finally, also the application layer needs some adaptations to enable efficient integration. As described in [3.3], web services are ideal for machine to machine communication. Unfortunately, the existing protocols, such as HTTP, used for web services are too heavy and verbose for constrained nodes. For this reason, a lightweight RESTful application protocol, Constrained Application Protocol (CoAP), has been proposed [3.4]. CoAP provides similar functionality for constrained networks as the one HTTP provides for conventional networks. CoAP uses PUT, POST, GET, and DELETE methods to communicate, update or remove resources hosted by the sensor/actuator nodes. To maximize the benefit of web services using CoAP, a number of extensions are proposed. The most relevant extension to our work is resource observation [3.5]. One of the application areas of Sensor/Actuator Networks is monitoring of different environmental phenomena. In such systems, sensors gather information and send it to a monitoring station so that the appropriate action can be taken. The communication between the sensor and the monitoring station can be done through

frequent polling, but results in several unnecessary request/response pairs if the values do not change frequently. By adding an observe mechanism, sensors can inform interested parties about any state changes of resources they want to observe. Communication can be optimized further by sending notification criteria while registering for observation. This way only changes that have significant importance to the observer are communicated. Details of this method, called Conditional Observation, can be found in [3.6]. All in all, CoAP and its extensions enable the interaction with constrained devices in a RESTful way over IP networks. This interaction can be utilized further to build IoT applications that make use of sensor data or steer actuators. This can be achieved by interacting with constrained devices within browsers [3.7] or services running in the cloud.

This paper focuses on novel enablers on top of CoAP to facilitate the creation and configuration of IoT applications with low creation and configuration overhead. The first step in building such IoT applications is to enable direct interactions between constrained devices without the need or continuous presence of an intermediary. This option is not yet available in CoAP and represents one of the contributions of this paper, namely binding sensors and actuators so that they can directly interact with each other, eliminating the need for external devices to continuously coordinate communication between them. We show how the CoAP protocol and the observe option (along with the conditional observe extension) can be used to create such direct interactions, also called bindings, between sensors and actuators in a flexible way. The interactions themselves are fully RESTful CoAP-based interactions, allowing anything to be bound to anything. This offers a lot more flexibility than other binding solutions presented so far. On top of that, we propose configurable, connectable and reusable building blocks with CoAP interfaces, called RESTlets that perform some processing of data at different levels. The processing can be done inside the constrained network, at the edge of the constrained network or in the Cloud. It acts as an enabler to build IoT applications that consist of modular processing steps, potentially distributed along the path between the sensors and the Cloud. The RESTlets can accept inputs and produce outputs after performing basic processing and can be configured through control parameters. To link sensors and actuators to RESTlets or to create chains of RESTlets, we build upon the binding concept. This contribution shows that creation of (part of) an IoT application can be reduced to linking together devices and processing blocks, demonstrating feasibility of the approach. In addition, we investigate the performance in terms of overhead and how this can be optimized by looking at different options such as location of RESTlets, in-network processing, etc. This paper is organized in seven sections. The underlying protocol, CoAP is discussed in more detail in the next section and related work will be discussed after that. The binding concept will be discussed in Section 4 followed by a discussion of the RESTlet concept. Section 6 shows the

implementation and evaluation of the two concepts. The paper ends by discussing main findings and giving concluding remarks.

3.2 Constrained Application Protocol (CoAP)

Recently, IETF approved the Constrained Application Protocol (CoAP) [3.4] as an open standard suitable for machine-to-machine communication or IoT interactions. As it implements a subset of the Representational State Transfer (REST) paradigm, it is referred to as the lightweight counterpart of HTTP. CoAP employs the same four methods, namely GET, PUT, POST and DELETE, as HTTP when sending requests from a client to a server. However, unlike HTTP, CoAP uses UDP as transport layer protocol in order to avoid the message overhead and extensive resource requirements of TCP. Reliability is provided through confirmable messages, allowing a client to specify whether a message should be acknowledged or not. CoAP client/server communication takes place in the same way as any REST-based communication. Clients send a request to a specific resource identified by a URI to retrieve the current resource representation or to modify it. The server then replies with the current representation or a status message. For instance, as shown in Figure 3-1a, if a client would like to receive the current light intensity in a room, it just sends a GET request to the specific resource representing the light intensity. Upon receipt, the sensor responds with the current value. Figure 3-1a shows this client/server interaction where the resource is represented by `/s/l` and the current value, which is 80 Lux, is sent back to the client. The responses could be piggybacked or separate. Piggybacked responses contain the data along with the acknowledgement to a confirmable request. However, if the data is not available during the request, for various reasons, the acknowledgement is sent alone and the response follows when the node is ready to send the data. The other methods may not require data to be sent back, but responses that indicate the outcome of message will be returned. Figure 3-1b shows, a client that sends PUT to a resource, represented by `/a/l`, on a device such as a Dimmer to change the light intensity to 60 Lux. After adjusting the value, the dimmer responds with a status code, stating that the value is changed.

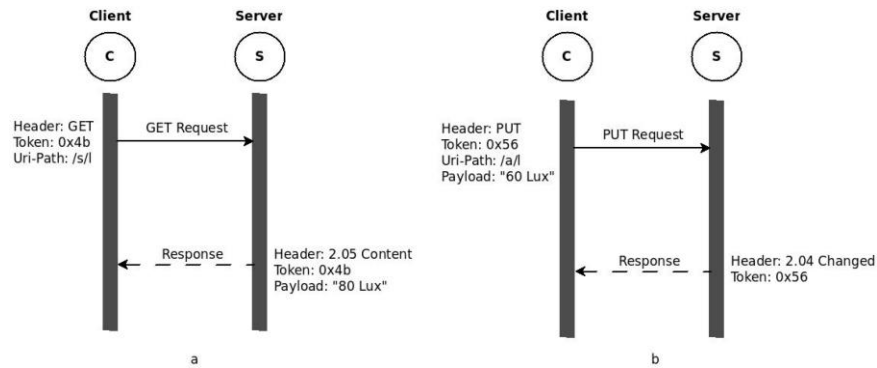


Figure 3-1: (a) CoAP GET Operation; (b) CoAP PUT Operation.

Responses can also be cached at intermediaries to improve efficiency. This feature is especially important for constrained devices as proxying will allow other devices to respond in place of the constrained node which might be sleeping or to limit traffic in the constrained network. CoAP requests and responses use a fixed four bytes header followed by zero or more compact binary options and an optional payload (Figure 3-2). The VER field (2 bits) in the header indicates CoAP version number and is always set to 1. The T field (2 bits) indicates message type. Message type 0 indicates CON (confirmable) request; and is used if reliability is required. Requests sent as CON must be acknowledged by setting the message type to ACK (value = 2). On the other hand, non-confirmable (NON) messages (Value 1), are used if reliability is not a requirement. Token length (TKL) field is part of the four bytes CoAP header and indicate the length of the token which immediately follows the fixed CoAP header. The Token is used to match requests with responses. Code (8 bits) field is used to indicate request method in requests and response code in responses. Message ID (16 bits) is used to detect duplicates and also match requests with responses. CoAP packets may contain one or more options to specify different aspects of the message such as URI and message format. Options are inserted in packets in ascending order and specified as option delta, length and value. If the message contains payload to be transmitted, an eight bit Payload Marker field is inserted followed by the payload. Refer to the CoAP RFC [3.4] for detailed explanation of the fields and CoAP operations.

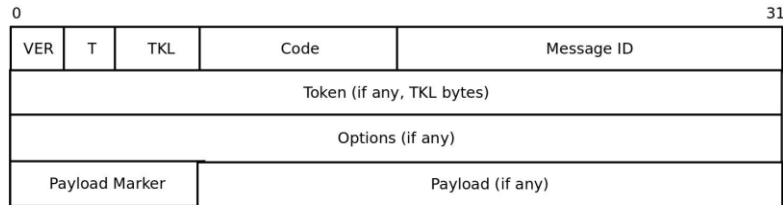


Figure 3-2: CoAP Header

As mentioned above, CoAP is designed following the REST architecture and is optimized for M2M communication in constrained environments. One of the application areas where M2M communication is widely used is resource monitoring. In such applications, clients need to have an up-to-date representation of data from servers. Polling (sending periodic requests to servers) is not optimal in constrained environments since the values may not change as often as the polling request frequency resulting in unnecessary packet transmissions. Introducing a mechanism that triggers transmissions only if changes occur improves the communication significantly. Resource Observation [3.5] is an interesting extension of the CoAP protocol that introduces such a mechanism. With the observe extension, clients can inform servers about their interest in getting an up-to-date representation of a resource by adding the observe option along their GET request. As a result, the server registers the client as an observer and sends the current representation. After that, the server only sends values, called notifications, when there is change in the resource representation. This method of communication is proposed based on the well-known observer design model. Figure 3-3 shows this communication model by taking the light sensor example discussed above. Instead of repeatedly requesting for the current representation, the client sends a GET with an observe option. This is called registration at the server. The server responds by including the current representation and registering the client as an observer. Subsequent changes in the resource representation will trigger notifications to be sent back to the client.

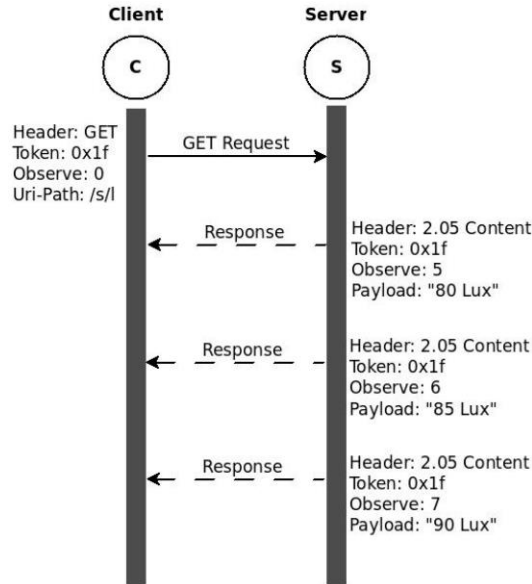


Figure 3-3: CoAP Observe Operation

In this communication model, the original request and subsequent notifications are matched using the Token. In frequently changing environments, the representations may arrive out of order at the client. The observe option value is incremented each time to identify the latest value. For a resource to be observed by clients, it must be defined as observable, also indicated by the obs attribute in the resource definition. A significant performance gain is obtained by using resource observation instead of polling. Nevertheless, not every event change might be significant enough for the client to store it or to trigger any action. Therefore, all those insignificant messages will be dropped after being transmitted over the constrained network. Those unnecessary transmissions can be suppressed by combining resource observation with server side filtering. Conditional Observation [3.6] provides a mechanism for clients to specify notification criteria during registration. As a result, servers will filter notifications and send only those that meet the notification criteria. Detailed implementation and evaluation of Conditional observation is given in [3.6]. The core-interfaces draft [3.8] also allows server side filtering by allowing clients to specify notification criteria in URI-Queries.

3.3 Related Work

3.3.1 Sensor-Actuator Interaction

The traditional approach for applications that require sensor-actuator interaction was to statically associate related devices at the time of deployment. This approach lacks flexibility. One of the earliest works that attempts flexible binding is the ZigBee End Device binding [3.9]. As stated in the specification, devices with similar End Device Profile and matching cluster ID can be bound. This dynamic binding method places many strict requirements on the devices that will be bound and, as a result, lacks the flexibility of binding any device with any other device that our solution provides. Another notable work that attempts to improve the limitation of ZigBee End Device Binding is given on [3.10]. This work avoids the requirement of matching cluster ID by matching sensor events with actuator actions. However, this solution is also based on ZigBee and devices that are not compatible with ZigBee cannot be included in the binding process and hence still lacks the flexibility we desire. To achieve maximum flexibility of binding any two devices, working on open standards is preferable. In line with this, the CoRE Interfaces draft [3.8] specifies how CoAP methods can be used to achieve flexible binding. The mechanism proposed in the draft allows end devices to establish a binding relationship between two resources through discovery mechanisms or through human intervention and then synchronizes the content of the involved resources. This solution has its advantages as it provides a generic solution that can be used in interface descriptions. However, the solution focuses on synchronizing the contents of two resources on different end devices. It is not possible to execute a specific action on the other device. Additional programming logic is still required to send the appropriate trigger to the same or different actuator.

3.3.2 In-Network Processing

Different developers have suggested different IoT application development models. Some prefer WS-* such as SOAP using HTTP while others argue that RESTful approaches are better suited for IoT [3.11]. A survey conducted among developers [3.12] concluded that RESTful web services were the preferred choice of most developers. However, even RESTful IoT applications have different development approaches. Traditional applications have been running at the edge of the constrained network or on the gateway. In recent years, many applications are moving into the cloud [3.13]. Actinium [3.13] is one such solution. Actinium divides the whole IoT application into Thin Servers that provide hardware functionality through RESTful interfaces and scripted apps, which run in the cloud and implement the IoT application logic. This allows developers to focus on programming their application to run on the cloud without dealing with the

constrained environment. This approach is significantly different from our approach which attempts to do as much processing as possible inside the LLN e.g., in order to reduce latency, to limit the amount of data going to the Cloud or to remain operational in the absence of connectivity. There are other initiatives, similar to ours, which attempt to keep some or all of the processing logic inside the LLN. Ref. [3.14] presents a programming abstraction known as T-Res which models processing tasks as resources that sit on a constrained device and can be manipulated by CoAP methods. Each T-Res resource stores URIs of the input and output devices as sub-resources. The last output and the compiled processing function (originally written in Python) are also stored as sub-resources. The processing function internally connects the input sources and output destinations by reading data from the input source(s) and sending out new outputs to devices identified by the URLs, if any. The last output is stored to allow concatenation of tasks. T-RES also provides getter and setter functions as programming APIs to be used in processing functions. Even though this system has some similarities to our solution, there are quite many significant differences, the first one being the overall approach. This solution represents processing tasks as resources while we model RESTlets to be independent IoT application building blocks that may run anywhere in the network (inside LLN, on Gateway or in the Cloud). We also store input that may arrive from any device and send stored output to any other device after processing, but take a different approach regarding the way processing is done. In case of T-Res, the processing function is responsible for getting the inputs from the sensors and sending the output, if any. Our solution separates input retrieval and task processing by using flexible binding for the interaction with sensors. Another difference, which is also a significant limitation of this solution, arises from the very architecture of the solution. Every task resource stores URLs of input sources and destination outputs. This means if the application requires doing the same processing task (e.g., Average) on different sets of sensors and/or sends output to different actuators, multiple task resources need to be defined and the same function will have to be stored in each resource.

Our system enables reuse of processing functionalities as long as the processing logic remains the same. In [3.15], virtual sensors and actuators are defined as resources to provide a mechanism to move part of the IoT application processing logic into the LLN. The virtual resources are defined hierarchically as template, instance and configuration resources. Whenever a new virtual resource is created on a device, the template must be posted on the virtual resource directory and the corresponding sub-resources. The input is pulled and calculation and generation of notification is done only when GET is issued by a component of the code in the cloud. Our solution interconnects the devices to talk to each other automatically without the need for external commands. Further, this solution still has its components in the cloud while ours tries to avoid putting the application code in

cloud as much as possible. LooCi [3.16] is another model for IoT applications. It uses an event-based binding model and standardized event types that allow easy component interactions and re-use of components. This approach uses Remote Protocol Call (RPC) for communication.

3.4 Flexible Direct Binding

In many IoT applications, sensors and actuators are deployed to work together. Examples of such applications include temperature sensors and thermostats, a light switch (sensor) and light bulb (actuator) in smart lighting solutions, and motion sensors and automatic door controls. Some old installations use static configurations where sensors are associated with fixed actuator(s) before or during deployment. This solution allows direct interaction between devices but has serious flexibility issues. If we need to change the association made during deployment, we need to reconfigure the devices all over again. Another solution often used in many applications is introducing an, often proprietary, intermediary between the sensor and actuator. In such solutions, the sensor sends the data to the intermediary device and the device sends the trigger to the actuator. This indirect communication involves too many unnecessary transmissions of packets between the three devices. In addition to this, every communication needs to move to the edge of the LLN (or even further into the cloud), resulting in delayed response due to higher latency. Moreover, if the intermediary device is down for any reason, the whole communication between the devices will not take place. In this section, we introduce the concept of flexible direct bindings, which solves the aforementioned problems by allowing direct interaction of smart objects without losing flexibility. To avoid vendor lock-in and allow devices from different vendors to communicate with each other, the implementation of this concept is realized as an extension of the CoAP protocol and Observe option. To illustrate this, we consider a simple smart lighting system that uses a light switch (the sensor) as a sensor, triggering a light bulb (the actuator) whenever pressed. This association may be made or modified at any time without the need for complete reconfiguration. We will use RESTful web services for the application and, represent the sensor resource by `/gpio/btn` and the actuator by `/lt/on` following the IPSO naming convention [3.17]. In traditional intermediary-based solutions, all activities are coordinated by an intermediary device. As shown in Figure 3-4a, the communication begins by establishing an observation relationship between the intermediary, labelled Initiator, and the sensor. After that, whenever there is an event that results in a state change of the button resource (i.e., whenever there is a button press), the sensor sends a notification to the intermediary. Upon receiving the notification, the intermediary triggers the actuator. This can be even applied to dimmers where the light intensity of the bulb is related to the value sent from the switch as payload

during notification. As can be seen in the figure, the intermediary must be always available for the system to work.

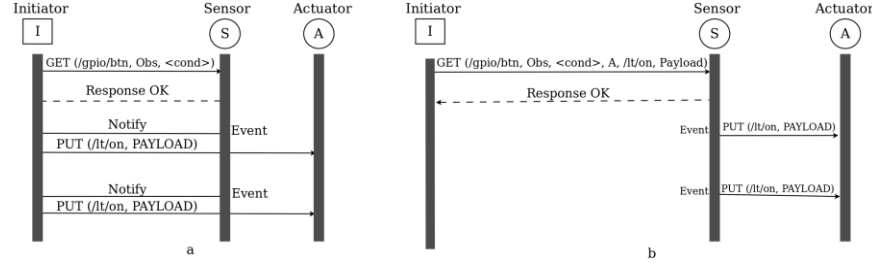


Figure 3-4: Sensor-Actuator Interaction. (a) Indirect; (b) Direct Binding.

On the other hand, using the flexible direct binding approach that we propose, the initiator is only required to establish the relationship between the sensor and the actuator. After successful establishment of the binding relationship, the initiator is no longer required to take part in any of the subsequent communication between the sensor and the actuator (Figure 3-4b). This means, the initiator could be any IP capable device, such as a smart phone, that will just establish a relationship and then leaves and re-enters the network anytime. This relationship exists as long as the sensor and actuator are functional or until we exclusively change the binding. The existing CoAP protocol does not support establishing binding relationship between two devices through a third device. When a device sends an observe request to another device, the relationship that will be established is between the sender and the receiver. A mechanism that allows the receiver to differentiate between a binding request and a traditional observation request needs to be introduced for our solution to work. In addition to this, some details of the actuator have to be communicated along with the binding request. In line with this, we modified the CoAP protocol by introducing four additional options, namely BIND_URI_HOST, BIND_URI_PORT, BIND_URI_PATH and BIND_PAYLOAD. BIND_URI_HOST and BIND_URI_PORT (optional in case default port is used) are defined to indicate the IP address and port number of the device that needs to be notified when events arrive. BIND_URI_PATH is the resource representation on the device through which it is triggered. When the request is stored on the sensor, the presence of these options is used to differentiate between a binding relationship and a traditional observation relationship. The optional BIND_PAYLOAD option may be used in a request if we wish a specific value to be sent during notification. In its absence, the new resource representation will be sent to the actuator. The method used for the notification will be PUT. Figure 3-5 shows the flow chart that describes how a binding relationship is established while Figure 3-6 shows the notification process. As shown in Figure 3-5, when a new request from the initiator that contains the observe option arrives at the sensor, it checks for the presence of one of the newly defined options to

identify whether the request is a binding request or not. If so, the information contained in the newly defined options will be used as details to define the observer. If not, the source IP address, source port number and source URI_PATH will be taken as IP address, port number and URI path of the observer. For binding requests, we may optionally store the address of the initiator too. Once this is done, the initiator will not be involved in any event notification communications.

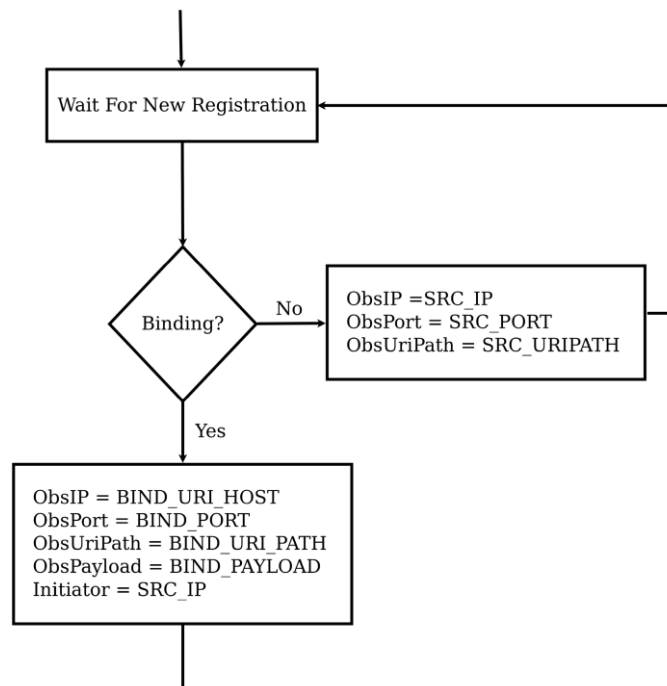


Figure 3-5: Flow Chart Showing Binding Relationship Establishment.

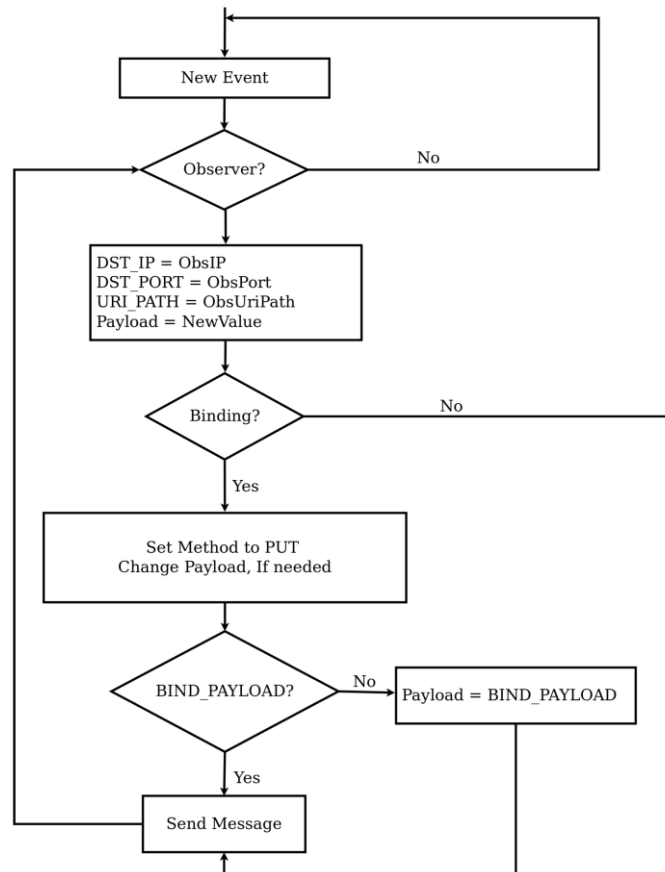


Figure 3-6: Flow Chart Showing Notification of Events.

Referring to Figure 3-6, event notification of both binding and observation relationship is conducted in a similar way. The notification process starts when the state of an observed resource changes, which causes a packet to be sent to all observers. The notification process is almost the same for both binding and observation relationships except two steps. The first difference is the method. In case of a normal observation, a response packet is created while in case of a binding a PUT message is prepared. The payload to be used for binding relationships can be modified by the BIND_PAYLOAD option, which is the second difference. Apart from these two differences the notification process is the same. Management of the binding relationships can be facilitated by introducing binding directories. Binding directories are similar to resource directories but list existing binding relationships. Through the binding directory, we may find out which binding relationships exist and perform reconfiguration by sending other binding requests, if necessary.

3.5 RESTlets

More and more IoT applications are moving into the Cloud. Many others reside at the gateway running proprietary protocols. Both options require movement of all data generated by sensors to the edge of the LLN or to the Cloud. As discussed in the previous section, this approach puts heavy burden on the border devices and some of the data that traverses the LLN might not be useful for the IoT applications. Performing some (pre) processing activities inside the constrained network can help to reduce the number of packets transmitted to the cloud or the edge. In this section we will introduce a novel solution that breaks down IoT applications into smaller and manageable units in order to simplify IoT application development and resolve the problems mentioned above. The solution is based on what we call RESTlets. RESTlets can be defined as IoT application building blocks that use RESTful web services to process data inside the constrained network. RESTlets have one or more data and control inputs, processing logic and outputs (Figure 3-7). Internal wiring connects the data inputs, control inputs and the processing logic to generate new outputs. In the context of sensor/actuator interactions the data inputs could be sensor readings or outputs of other RESTlets. The processing that will be done inside the RESTlet may vary depending on the requirement of the application. The processing could be as complex as sending an SMS or as simple as a logical AND. If the processing results in new data, the output could be used to trigger an actuator or sent to another RESTlet as an input. The control inputs are configuration parameters that can be used to control how the RESTlet operates and how or what outputs must be produced. For example, the control parameters may define the threshold value for generating new outputs, or even the computation interval. By modifying the control parameters during runtime, we can control how a specific RESTlet behaves. After a RESTlet is defined on a specific device, it can be instantiated multiple times.

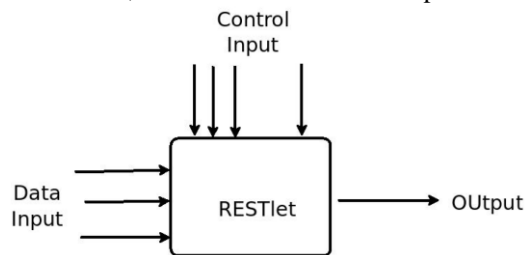


Figure 3-7: RESTlet Block Diagram.

Implementation of RESTlets using RESTful web services can be achieved by representing each RESTlet instance as a resource and each component (data input, control input and output) as sub-resources. Instantiation of RESTlets is achieved by sending POST requests to the device that is selected to host the RESTlet. Since multiple RESTlets can be defined on a single device, the POST request must

contain the name of the RESTlet along with the number of data inputs, control inputs and outputs. Whenever a POST request is received, the RESTlet resource and its sub-resources will be created dynamically and will be referenced using hierarchical naming convention as:

`/r/<RESTletNo>/<in|out|con>/<SubResourceNumber>`

For instance, `/r/0/in/0` refers to the first data input of the first RESTlet while `/r/1/out/0` and `/r/0/con/1` respectively refer to the first output of the second RESTlet and the second control input of the first RESTlet. Please note that RESTlets are numbered as per the sequence of the POST request. This means the first RESTlet is the one created by the first POST request and so on. Alternatively, we can name them by passing the RESTlet name with the request for creation. The combination of the RESTlet concept and the flexible direct binding concept can be used to build simple web-based IoT applications. To show how this works, we consider a simple smart home application that turns on or off the air conditioner based on the average temperature in an occupied room. The temperature values are obtained from three temperature sensors and occupancy of the room can be identified by a motion sensor. In traditional Web-based IoT applications, the data from the sensors is sent to the LLN gateway to be processed there and the result will be sent back into the LLN for the actuator (in this case attached to the air conditioner) to act upon it. The data transmission from the sensors takes place in various ways. One possible way is by periodically polling for new values (shown in Figure 3-8). This would result in a lot of unnecessary data transfers in case the frequency at which temperature values change is much lower than the frequency of the polling interval. Another option would be to establish an observation relationship between the gateway and each sensor so that only new changes are communicated to the gateway. In both cases, data from all sensors is transferred to the gateway and triggers for the actuator, if any, have to be sent back into the LLN.

```
While (1)
{
    Temp1 = getTemperature(S1);
    Temp2 = getTemperature(S2);
    Temp3 = getTemperature(S3);
    Motion = getMotion(M);
    AverageTemperature = (Temp1 + Temp2 + Temp3)/3;

    If (AverageTemperature > 25 AND Motion) then
    {
        NotifyAirco (ON);
    }
    elseif (!Motion) then
    {
        NotifyAirco(Off);
    }
    Sleep();
}
```

Figure 3-8: Sample Code Executed on Non-constrained Devices

Using RESTlets we can move some or all of the processing logic inside the LLN to reduce the number of packets transmitted. A block diagram showing the breakdown of the application into RESTlets is given in the diagram (Figure 3-9). The three temperature sensors send their data to the RESTlet which implements the AVERAGE function. The output of this RESTlet is sent to another RESTlet which implements the logical AND operator that combines this with the readings of the motion sensor to finally send the trigger to the actuator. One of the benefits of the RESTlet approach is that, any of the existing sensor or actuator nodes can be used to host the RESTlets. Alternatively, we can distribute the RESTlets among different nodes. Yet another alternative may be placing a more capable node inside the LLN that does all the processing. For simplicity, let's select node S1 to host the AVERAGE RESTlet and the motion sensor, M, to host the AND RESTlet. The figure below (Figure 3-10) shows the nine steps that can be used to program this application.

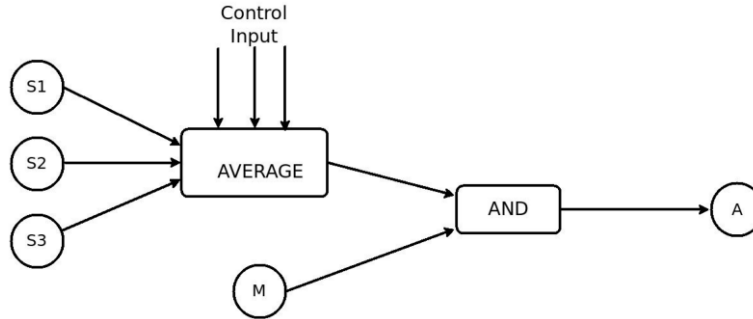


Figure 3-9: RESTlet block diagram for the smart home scenario.

```

1. POST [S1]/r;3,1,1
2. POST [M]/r;2,1,1
3. PUT [S1]/r/0/c/0;25
4. GET [S1]/s/t;obs:[S1]/r/0/in/0
5. GET [S2]/s/t;obs:[S1]/r/0/in/1
6. GET [S3]/s/t;obs:[S1]/r/0/in/2
7. GET [M]/s/m;obs:[M]/r/0/in/0
8. GET [S1]/r/0/out/0;obs:[M]/r/0/in/1
9. GET [M]/r/0/out/0;obs:[A]/a/t

```

Figure 3-10: CoAP Messages used to create the required Binding Relationship

As shown in the above listing, the first two messages create the AVERAGE and the AND RESTlets on nodes S1 and M, respectively. The numbers indicate the number of data inputs, control inputs and output in that order. In this case, the AVERAGE RESTlet contains 3 data inputs while the AND RESTlet only has two. Both RESTlets have one control and one output each. The third statement sets the control parameter of the AVERAGE RESTlet to be 25 so that only average values greater than 25°C are sent as output. Statements 4 through 6 establish binding relationships between each temperature sensor and the three data inputs of the AVERAGE RESTlet. After receiving these messages, the sensors send all temperature changes to the RESTlets data inputs. The seventh statement associates the motion sensor to the first input of the AND RESTlet by establishing a binding relationship between them. The output of the AVERAGE RESTlet and the second data input of the AND RESTlet are conveniently associated through observation relationship by the eighth statement. The last statement finally associates the output of the AND RESTlet to the actuator so that changes at the output will trigger the actuator. For the binding process to work properly, all outputs of RESTlets are

made observable. Interestingly, this simple concept reduces the whole IoT application development to a series of CoAP message transmissions that may be sent from anywhere in the network or over the Internet. Management of the IoT applications is also made easy. Sending simple GET messages to the RESTlet nodes and using binding directories to list out all available bindings gives us enough information to inspect and, if needed, reprogram the entire application or to modify some aspect of it.

3.6 Implementation and Evaluation

3.6.1 Implementation

The selection of a good implementation platform is crucial to demonstrate the feasibility of new concepts and to show performance gains obtained through the proposed solutions. We used Contiki2.7 [3.18] as a base system for all experiments, which was the latest stable version available at the time of starting our experiments. Contiki is an open-source embedded operating system suitable for constrained systems. Its innovative IP implementation, uIP, makes it a good solution for experiments that involve IP-based communications in the constrained world. In addition to this, all required features for our tests such as 6LoWPAN, RPL, and CoAP have all been implemented. The CoAP implementation of Contiki is known as Erbium [3.19]. Next to this, we also need to run some tests on non-constrained devices for comparison purposes. This requires a CoAP implementation that runs on non-constrained devices (e.g., gateways, Cloud servers, etc.). For this, we have used our own C++ based implementation of CoAP and its extensions, named CoAP++. Since both Erbium and CoAP++ do not support the proposed binding and RESTlet concepts, some modifications have been made, including the addition of new CoAP options and the introduction of a mechanism to define and instantiate RESTlets.

3.6.1.a Flexible Direct Binding

As explained earlier, to support the binding concept, four new options were introduced. These new options have been added to Erbium and CoAP++ with the following option numbers 42, 46, 50, and 54 for BIND_URI_HOST, BIND_URI_PORT, BIND_URI_PATH and BIND_PAYLOAD respectively. In addition to this, the required functionality for serializing and parsing those options has been added. Apart from this, two other modifications have been made in order for the binding solution to work properly. The first major modification included an extension of the registration mechanism in order to differentiate between normal observers and binding observers as shown Figure 3-5. The second major change was the way notifications were sent to observers, as for binding relationships, the

PUT method is being used, optionally in combination with a payload as indicated by the BIND_PAYLOAD option.

3.6.1.b RESTlets

The RESTlet concept, as discussed above, makes use of bindings to build (parts of) IoT applications by performing processing tasks and exchanging raw values and (semi)processed data between devices. In order to enable nodes to support RESTlets, some modifications have been made to both Erbiu and CoAP++. The main modifications are discussed below. The application logic of RESTlet, which acts on the data inputs (and the control inputs) to produce outputs, has been implemented for every node that potentially hosts the RESTlet. Every RESTlet defines its own processing function and hence one processing function per RESTlet is defined. Generic functions that manipulate the data and control inputs have also been defined. A POST request to a specific resource initiates the instantiation of the RESTlet instances and the dynamic creation of associated resources. The /r resource is defined for this purpose. Moreover, since Erbiu does not support the dynamic creation of resources, this functionality has been added. In order to differentiate one RESTlet instance from the other, detailed information about every instantiated RESTlet needs to be stored. Therefore, a data structure for storing the RESTlet name, the number of data inputs, the number of control inputs, the number of outputs, and the memory address of the processing function has been defined. This data structure may also store the latest values of the data inputs, control inputs and outputs if required. Finally, a callback function that is called for further manipulation of the resources and sub-resources has been defined. The flow chart of this callback function is shown in Figure 3-11. Each time a request for the /r resource arrives, it is forwarded to the callback function in order to see which sub-resource is referenced and an appropriate action is taken. Based on the URI-PATH of the request, the function will be able to identify for which sub-resource the request has been sent. A PUT request to the data input (in) initiates execution of the processing logic, which, in turn, notifies observers in case the output changes. Sending a PUT request to the control input (con) results in changing the identified control input. GET requests to any of the sub-resources may be used to retrieve the current value. All outputs have been made observable in order to allow binding relationship between the RESTlet and devices (or other RESTlets).

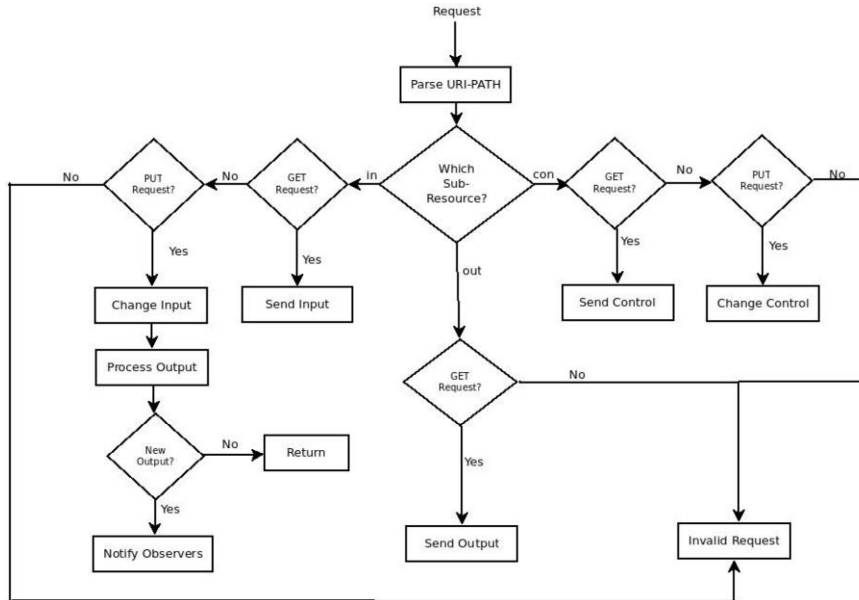


Figure 3-11: Flowchart Showing Interaction with RESTlet Instances using CoAP Messages

Putting it all together, once all logic for a particular RESTlet has been defined and implemented, applications may instantiate RESTlets on a specific node by sending a POST request to the `/r` resource and specifying the name of the RESTlet, the number of data inputs, control inputs and outputs in the payload as follows:

RN = <RESTlet Name>; IN = <# Data inp>; CON = <# Control inp.>; OUT = <# Output>

For example, a POST request sent to a node with `RN=AND;IN=2;CON=2;OUT=1`; as payload, creates the AND RESTlet with two data input, two control input and out output resources. The five resources will be referenced as `/r/0/in/0`, `/r/0/in/1`, `/r/0/con/0`, `/r/0/con/1`, and `/r/0/out/0` in all further communications. Subsequent POST requests create RESTlet resources that are identified by changing the number next to the `/r`.

3.6.2 Experiment Setup

For all experiments involving constrained nodes, we simulated Zolertia (Z1) nodes in Cooja. The basic scenario we tried to simulate is the interaction between a temperature sensor (as sensor), identified by the `/s/temp` resource, and a thermostat (as actuator), identified by `/a/t`. The temperature values are periodically read from a random sequence of 100 values stored in an array. If two consecutive readings

result in different values, a notification to the observers will be sent. Whenever a non-constrained node is involved, we use the CoAP++ code running on the laptop.

3.6.3 Functional Evaluation

In this subsection the details of the proposed solutions and their implementation are discussed.

3.6.3.a Bindings

As explained earlier, binding a sensor and an actuator can be done by any device from anywhere in the Internet. Figure 3-12 shows the CoAP++ GUI screenshot when a GET request is used to establish a binding between the /gpio/btn resource on a sensor with IP address [aaaa::c30c:0:0:2] and an actuator with address [aaaa::c30c:0:0:3]. The specific resource of interest on the actuator is /a/t.

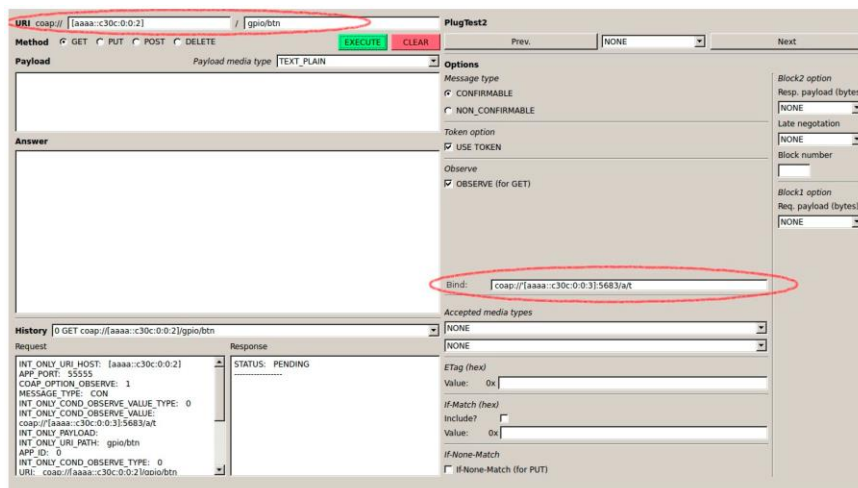


Figure 3-12: Creation of Binding Using CoAP++ GUI from Non-constrained Device

Once the binding relationship has been established, all further interactions take place directly between the sensor and the actuator. One such an interaction can be seen in Figure 3-13, which is Screenshot of the Simulation in Cooja. The Cooja Visualizer at the left of the picture shows the direct communication (the notification and the ACK) in blue arrows. The shaded part of the simulation script editor window confirms the route the packets followed after data is generated at the sensor (Node 2) until received by the actuator.

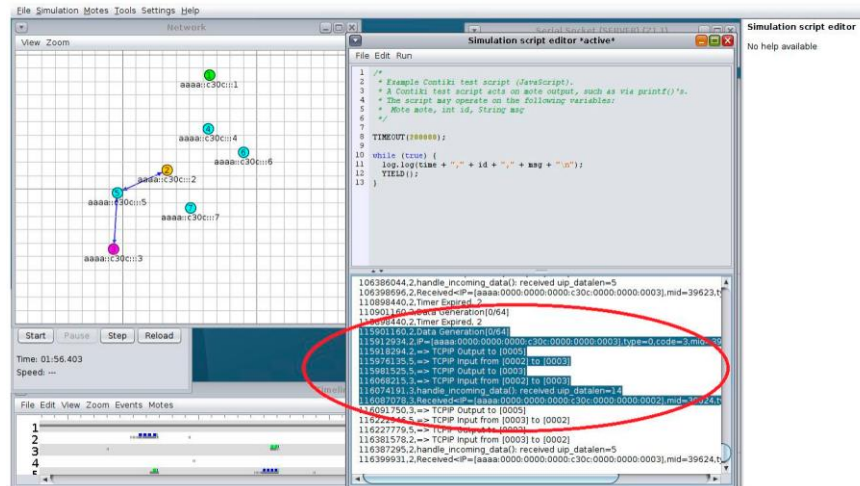


Figure 3-13: Direct Interaction of Sensor and Actuator Nodes in Cooja

3.6.3.b RESTlets

RESTlets are application building blocks that may be defined once on devices and that can be instantiated a number of times to build (part of) IoT applications by interconnecting them with devices and each other using flexible bindings. This process can also be accomplished using any device connected to the Internet. To illustrate this, we consider the example of a simple LESS-THAN RESTlet in order to send notifications to the actuator shown above in case the temperature drops below 25 degrees. Figure 3-14 shows the Copper screenshot of this operation. Sending a POST request to the `/r` resource of the node selected to host the RESTlet, in this case `[aaaa::1]`, creates the resource and its sub-resources. The payload shows the name of the RESTlet, LT indicating less than, which has one data input and one control input. The control input is initialized to 25. The default number of outputs is 1. Upon reception of the POST request the RESTlet is being instantiated and further referenced as `/r/0`. In addition, 3 sub-resources identified as `/r/0/con/0`, `/r/0/in/0` and `/r/0/out/0` are created representing the control input, data input and output, respectively.

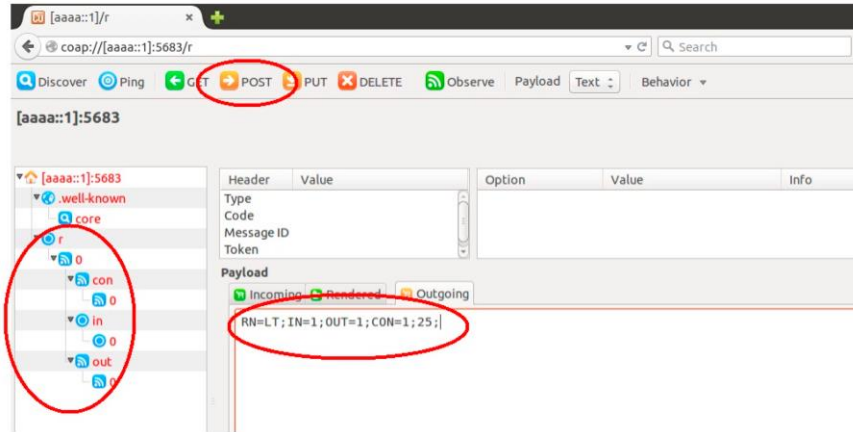


Figure 3-14: Creation of RESTlet Instances in Copper

To achieve the desired result, the input `/r/0/in/0` is bound to the sensor node and the output `/r/0/out/0` is bound to the actuator. This way, sensor readings are being transmitted to the RESTlet and the output of the RESTlet triggers the actuator. For this, the binding functionality shown in the previous sub-section is applied twice. Subsequent updates of the control parameter can be easily performed by sending a PUT request to the `/r/0/con/0` resource. This solution can be used to build complex IoT applications by distributing the RESTlets at different devices inside the LLN, at the LLN Gateway or even in the cloud. Irrespective of the complexity of the application or the location of the RESTlet nodes, we send a series of CoAP requests to the devices to program the application. To simplify the development even further, we can employ visual programming tools to simply drag-and-drop components to instantiate RESTlets and perform the binding.

3.6.4 Performance Evaluation

The proposed modifications may affect some aspects of the network or the device itself. Memory footprint, number and size of packets transmitted, and communication delay are some of the parameters that might be affected positively or negatively. The outcomes of several tests showing these impacts are discussed below.

3.6.4.a Performance Evaluation of Bindings

A. Memory Footprint

As described above the original Erbium code has been modified in order to support the binding concept. This modification induces a slight increase in memory space, mainly in the code (text) segment. Table 3-1 shows the increased memory footprint of the binding solution compared to gateway or cloud-based solutions. As every observer's information needs to be stored in memory, the memory required in the

BSS section increases proportionally to the number of observers. However, the difference between the two approaches is only 38 bytes per observer for the two cases.

Table 3-1: Memory Foot Print (Byte).

Num. Observers	Binding Sensor				Non-Binding Sensor			
	Text	Data	BSS	Total	Text	Data	BSS	Total
0	47,829	306	5324	53,459	46,453	306	5166	51,925
1	47,829	306	5580	53,715	46,453	306	5384	52,143
2	47,829	306	5836	53,971	46,453	306	5602	52,361
3	47,829	306	6092	54,227	46,453	306	5820	52,581
4	47,829	306	6348	54,483	46,453	306	6058	52,797

Despite the slight increase in memory footprint, the code can still fit in constrained devices. Given the advantage of the binding solution, the increase in memory footprint is acceptable and the binding solution is viable to be applied in constrained devices. However, this does not come without a limitation. An increased number of bindings leads to an increase in memory space requirement. The BSS section of both solutions in the table shows that when the number of observers increases, the size of the BSS increases as well because at boot time the program always reserves the maximum amount of memory needed to store all potential observers. As memory is a very scarce resource of constrained devices, this will limit the number of observers allowed to register simultaneously and thus the number of bindings that can be supported. Here the gateway/cloud solution has an advantage since it may achieve scalability by aggregating multiple observe requests at the gateway avoiding one to one relationships between multiple actuators and a sensor.

B. Packet Size

LLNs have a low Maximum Transmission Unit (MTU). Large packets whose size exceeds the MTU go through a fragmentation/defragmentation cycle from the source all the way to their final destination. This behaviour negatively affects the performance of the network. Therefore, the resulting packet size is a very important parameter when discussing the performance of new solutions. Moreover, fragmentation also comes at the expense of an increased delay. The packet size at the application layer for CoAP based communication can be calculated as:

$$\text{PacketSize} = \text{Sizeof}(\text{CoAP-Header}) + \text{Sizeof}(\text{Token}) + \text{Sizeof}(\text{options}) + \text{Sizeof}(\text{payload})$$

where:

$$\text{Sizeof}(\text{CoAP-Header}) = 4\text{bytes}, \text{Sizeof}(\text{Token}) = 0 \text{ to } 8 \text{ bytes}$$

Size of (Options) differs from packet to packet depending on the number and type of CoAP options being included in the packet. For example, Observation requests include the Observe Option, which has a maximum length of four bytes. The Uri-Path option and payload greatly vary depending on the resource identifier and the data to be communicated. For the URI path, we assume the simplified IPSO Application Framework [3.17] resource names. For instance, for a button associated with a light switch (sensor) the URI path becomes /gpio/btn, which will be transmitted as two Uri-Path options with a total length of nine bytes (one byte for every option plus the length of both segments “gpio” and “btn” in the URI). Most of these values are common for all types of communication so they do not impact the comparison between the two methods. The real difference between the two solutions can be seen at the relationship initiation packet. In case of the non-binding solution the options that are minimally needed are Observe (one byte) and Uri-Path (nine bytes for /gpio/btn). Including the CoAP header (four bytes) and the token (one byte in this example), the total packet size will be 15 bytes. However, for direct bindings, the initial packet includes four additional binding options containing the information on how to trigger the actuator. Therefore, the number of additional bytes required, B_{Byte} is given by:

$$B_{Byte} = \text{Sizeof}(\text{BIND_URI_HOST}) + \text{Sizeof}(\text{BIND_URI_PORT}) + \text{Sizeof}(\text{BIND_URI_PATH}) + \text{Size of}(\text{BIND_PAYLOAD})$$

where:

$$\text{Sizeof}(\text{BIND_URI_HOST}) = O + 16 /*IPv6address*/$$

$$\text{Sizeof}(\text{BIND_URI_PORT}) = O + 2 /*Optional. Default CoAP Server port is used*/$$

$$\text{Sizeof}(\text{BIND_URI_PATH}) = \text{Sum of} (O + \text{size of} (\text{path_segment } i))$$

with i going from 1 to # of path segments:

$$\text{Sizeof}(\text{BIND_PAYLOAD}) = O + X$$

In the above formula, O is the number of bytes needed for encoding the option delta and option length (between one and five bytes, but one in most cases). The value X depends on what we want to transmit in the payload. In our example, we send a single byte information and hence X is equal to 1. Further, we assume the actuator uses the default CoAP server port. Using this formula, the additional number of bytes required for our example is given by $B_{Byte} = 19 + 6 + 2 = 27$ Bytes. Considering the 15 common bytes, the total packet size for the binding solution will be 42 bytes. Even if the packet size of the binding solution is bigger than the one of the gateway-based solution, it does not affect the network performance at all. First, this request is sent only once in order to establish the relationship. Once the binding has been established, there is no further communication of this size. Had it been the packet size of the notification, it would, indeed, impact the network negatively. In addition, the packet size is yet in the limit of the LLNs MTU, being

127 bytes at the MAC layer. Hence, no fragmentation will be applied that negatively affects the network performance.

C. Communication Delay (Latency)

Delay is an important parameter to compare the performance of different solutions. The route packets take to reach destination plays an important role in determining the communication delay. The route, in turn, depends on the network topology. Therefore, we need to consider different topologies to compare latencies between the two approaches. In this experiment, we considered four topologies as shown in Figure 3-15.

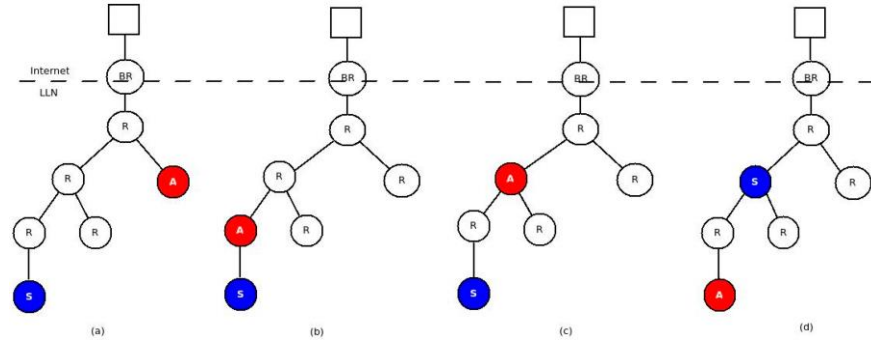


Figure 3-15: Topologies: (a) Sensor and actuator in different branch of the tree; (b) Actuator between Sensor and Gateway—directly connected; (c) Actuator between Sensor and Gateway after 1 hop; (d) Sensor between Actuator and Gateway after 1 hop.

For all topologies, the latency is computed as the time difference between the occurrence of the event at the sensor and the reception of the PUT packet by the actuator. From the results depicted in Figure 3-16, we can see that in all cases the gateway/cloud based solution has a significantly higher latency compared to the binding solution. This is expected as all sensor events are sent all the way to the gateway and triggers come down to the actuator in the non-binding solution. This increased number of hops introduces significant delay in the overall notification/trigger cycle. The delay will be even more pronounced for larger networks.

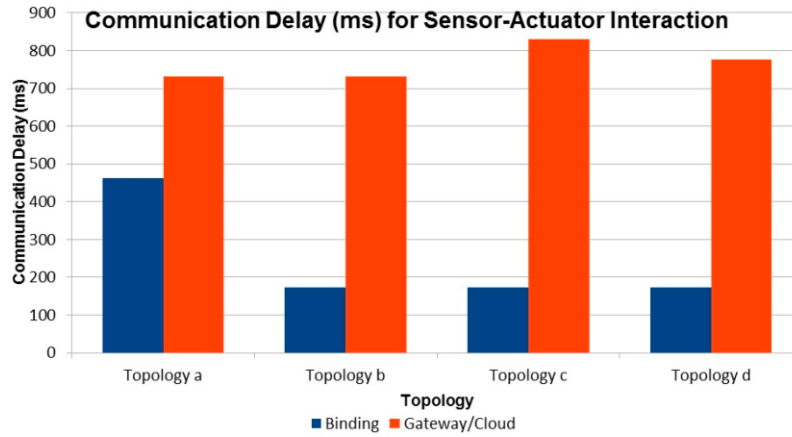


Figure 3-16: Communication Delay (ms) vs. Topology

For our solution, the number of hops, and hence the delay, depends on the routing protocol. As we mentioned earlier, we used RPL as routing protocol. In RPL, the furthest the packets travel is until the common parent of the sensor and actuator. The closer the sensor and actuator, the less delay is introduced. The Cooja screenshots (Figure 3-17) confirms this statement. The blue arrows in the left of Figure 3-17a show that the interaction is direct between node 2 and 3 while that of Figure 3-17b shows that the interaction goes through the border router. The shaded part in the right shows the route the packets take from the sensor to its ultimate destination (the actuator).

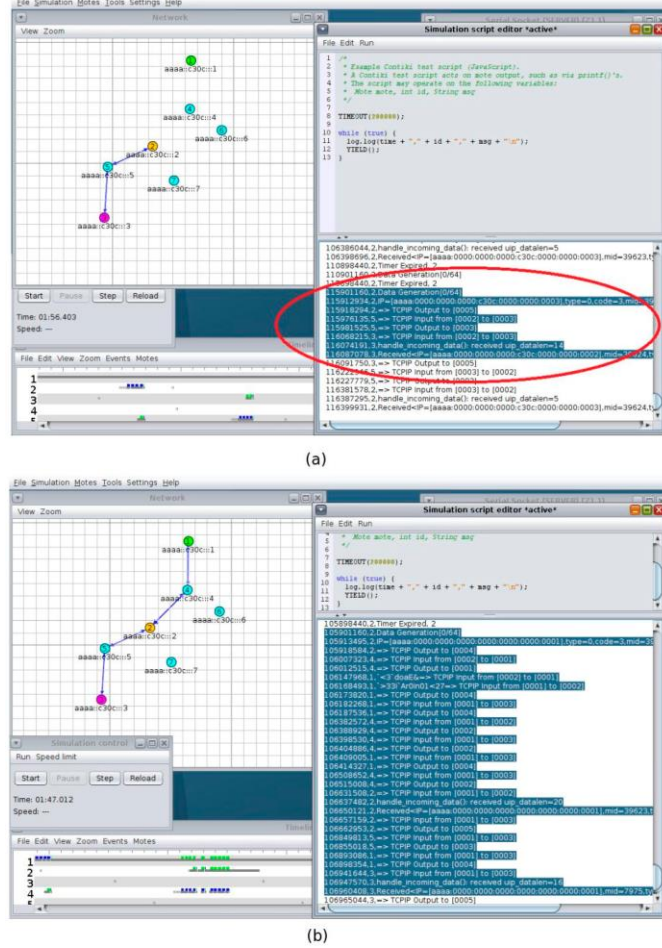


Figure 3-17: Sensor-Actuator Interactions. (a) Binding (b) Gateway/Cloud-Based Solution.

D. Number of Packets

An increased number of packets in constrained networks lead to an increased power consumption at each router node and more delay. Therefore, looking at the number of packets generated by the two solutions that strive to achieve the same goal is a good performance measure to compare both solutions. As every notification goes through the gateway, the gateway-based solution creates one additional packet for every notification. If the packets are sent as confirmable requests, this number will be doubled. As the number of sensors and actuators increases, the number of packets generated will also increase significantly. In dynamic systems where notifications are generated frequently, the number of packets being generated gets higher and higher.

3.6.4.b Performance Evaluation of RESTlet

Several tests were conducted to evaluate the performance of RESTlets. In all tests, we considered latency to be the most important performance factor that needs to be compared. We used different topologies, number of nodes, and data processing entities.

A. Impact of RESTlets

Data processing performed in the LLN by RESTlets introduces delay but reduces the number of packets in the network. We used a fixed topology (Figure 3-18) to mathematically evaluate the impact of RESTlets at the RESTlet node (labeled RN in the figure). In this scenario, data may be sent from the sensor nodes, labeled S, and pass through the RESTlet node before going out to the LLN gateway or the cloud. We compared the RESTlet case, where processing is done by the RESTlet node and No-RESTlet case, where the processing is done elsewhere (at the gateway or in the cloud). In the No-RESTlet case, node RN is used as a router only. Whenever a packet arrives, it just processes it in order to determine the next hop address after which it is forwarded to the next hop.

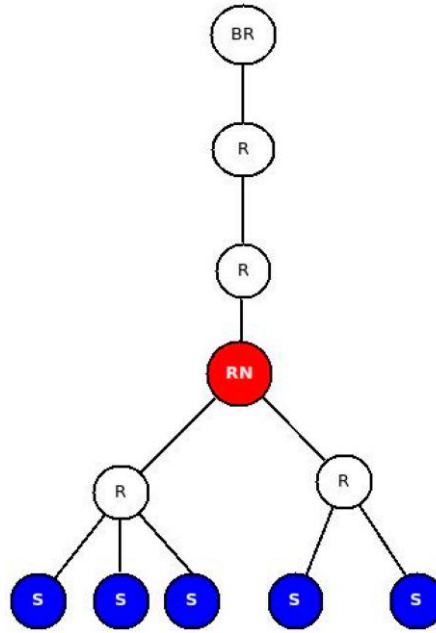


Figure 3-18: Network Topology

Therefore, in the No-RESTlet case, the total packet processing and forwarding time at the node RN is given by:

$$T_p = T_x + TF_1 + (TD_1 + T_x + TF_2) + \dots + (TD_n + T_x)$$

where:

- T_p is total packet processing and forwarding time.
- T_x is packet processing time (from the experiment we found out that this value is 6 ms)
- TD_i is time delta between the arrival of two consecutive data packets from two different senders (if only 1 data sender, this value is 0). This value is variable.
- TF_i is Packet forwarding time (calculated as the arrival time of the packet at the next hop minus the time the packet was ready to be sent out). This value is also variable.

So for n data generating nodes:

$$T_p = \begin{cases} T_x + TF_0, & n = 1 \\ T_x \times n + \sum_{i=2}^n TD_i + \sum_{i=1}^n TF_i, & n > 1 \end{cases}$$

On the other hand, in the RESTlet case, the node is expected to do other processing too. First of all, it has to unpack the CoAP packet to get the data and store it provisionally. Secondly, it waits for subsequent packets if more than one data sender node exists. Thirdly, it has to perform processing and generate output. Finally, a new packet is generated and forwarded to the next hop. Therefore, the total packet processing and forwarding time at the RESTlet node, T_p , is given by:

$$T_p = T_x + (TD_1 + T_x) + \dots + (TD_n + T_x) + T_{NPG} + TF$$

where:

- T_p is total packet processing and forwarding time.
- T_x is packet processing time. (From the experiment, we found out that this value is 22 ms and the RESTlet function we considered was AVERAGE. Other processing functions may yield different results).
- TD_i is time delta between arrivals of two consecutive data packets from two different senders (if only 1 data sender, this value is 0). This value is variable.
- TF is packet forwarding time (calculated as the arrival time of the packet at the next hop minus the time the packet was ready to be sent out). This value is also variable.
- T_{NPG} is time required to generate new packet (from the experiment we found out that this value is 14 ms).

So, for n data generating nodes,

$$T_p = \begin{cases} T_x + T_{NPG} + TF, & n = 1 \\ T_x \times n + \sum_{i=2}^n TD_i + T_{NPG} + TF, & n > 1 \end{cases}$$

In order to compare both results, we can say that TD_i is the same for both cases and can use an average constant number for simplicity. However, the value of TF_i is different among different packet transmissions. From the experiments, we observed that the data forwarding interval ranges between 20 ms and 140 ms. Moreover, all experiments showed that the processing time at node RN, T_x , is 6ms and 22ms for the No-RESTlet case and the RESTlet case, respectively. The new packet generation time, T_{NPG} , for the RESTlet case was also found to be 14ms. In order to see the difference in terms of processing time, we used the following simplified formula by using the aforementioned values as an average:

For No-RESTlet case (for n number of data generating nodes):

$$T_p = 6ms \times n + TD + (TF_i \times n)$$

For RESTlet case (for n number of data generating nodes):

$$T_p = 22ms \times n + TD + 14ms + TF_i$$

This enables us to calculate the resulted packet processing time for a varying number of data generating nodes (1, 2, 3, 4 and 5). Figure 3-19 depicts the result in graphs for both approaches using different TF_i values.

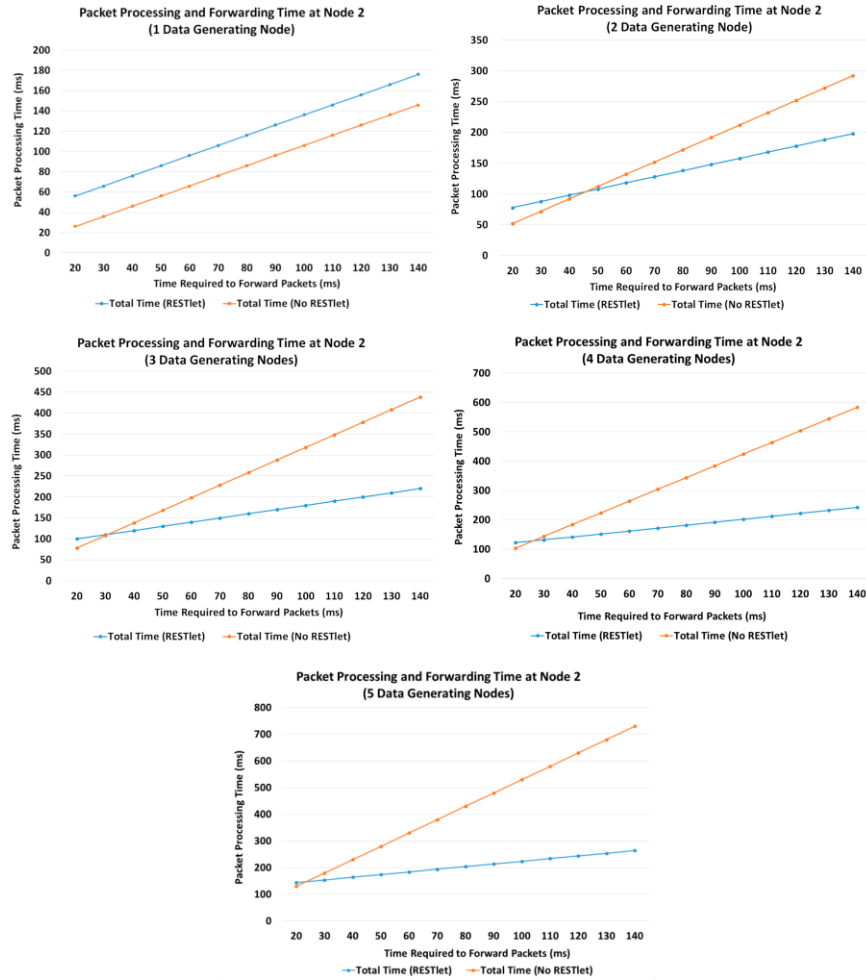


Figure 3-19: Packet Processing and Forwarding time at RESTlet Node for Various Number of Data Generating Nodes

Figure 3-19 shows that when the number of data generating nodes becomes more than one, the delay introduced by processing incoming packets by RESTlets becomes less important. For congested networks, which are characterized by larger TF_i values, the advantage will become more pronounced. This is due to the fact that the RESTlets only generate a single packet after processing (or no packet at all depending on the type of processing) whereas the No-RESTlet case blindly forwards all the packets it receives which will be subject to large forwarding times.

B. End to End Latency with Multiple Data Nodes (Impact of Number of Nodes)

In the previous subsection we mathematically showed that the reduced number of packets that results from the aggregation process by RESTlets compensates for the processing delay introduced by the RESTlets and result in better latency. To prove this concept, we measured the actual end-to-end delay from data generating nodes all the way to the border router. We used the topology shown in Figure 3-18 (above). The data nodes generate data every five seconds that will go to the border router. In the RESTlet case, all data is sent to the RESTlet node (RN) which processes the data and generates a new packet destined to the border router. The new packet is generated either upon arrival of data from all data nodes or within five seconds interval, depending on which condition is met first. The end-to-end latency is calculated as the difference between the data generation time of the first node and the arrival of the new packet at the border router. On the other hand, for the No-RESTlet case, all data is sent directly to the border router by traversing the RESTlet node as a router. In this case, the end-to-end latency is computed by taking the difference between the data generation time of the first data node and the arrival time of the last data packet at the border router. We run the tests by sending the packets as CONfirmable and NON-confirmable requests. Figure 3-20 shows the results.

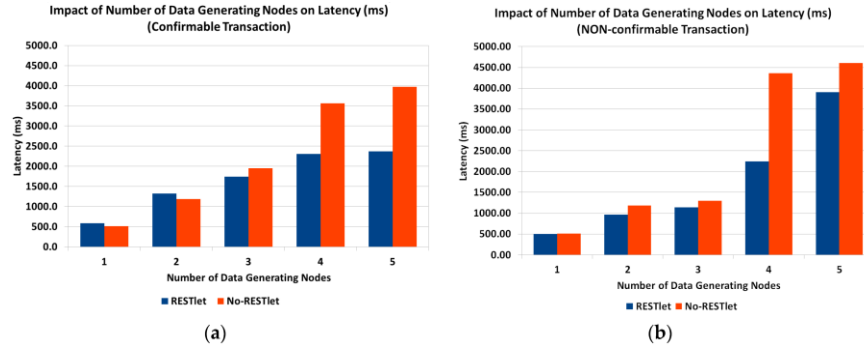


Figure 3-20: Impact of Number of Data Generating Nodes on End-to-End Latency. (a) Confirmable Communication; (b) NON-Confirmable Communication.

For both CON and NON transactions, the RESTlet case results in a reduced latency compared to the No-RESTlet case. The difference is significantly higher for the confirmable case when the number of data nodes is higher. In the no-RESTlet case, all data packets are forwarded to the border router which is expected to produce ACKs for all. This results in an increased load on the border router and hence increased latency. Even if the processing is done by an external more powerful device such as the gateway, still all requests, acknowledgements and responses have to go through the border router and contribute to the increased overall latency of the NON-RESTlet case. In the experiments we conducted, we made two interesting observations. When the transaction is CON, there were a number of duplicate packets and out of order arrivals especially when the data

nodes are more than three. This is much more visible for the NO-RESTlet case where, out of 75 packets sent, there were 28 duplicate packets while for the RESTlet case there were only 13 duplicates when the number of data generating nodes is five. The other observation is the difference in packet loss between NON and CON transactions. As expected, the NON transactions suffer from packet loss in both RESTlet and NO-RESTlet cases with staggering 15% and 30% loss, respectively when the number of data generating nodes is four. Finally, comparing the CONFIRMABLE and NON-CONFIRMABLE transmissions of packets, it is not surprising to see that the CONFIRMABLE messages result in a higher latency as compared to NON-CONFIRMABLE transactions. However, when there is large number of data generating nodes, the latency difference gets smaller for NON-CONFIRMABLE transactions. The reason is the higher rate of packet loss forces the processing of packets to be made at the end of the five second interval.

C. Impact of Other Nodes

Under normal working conditions, other communications may take place inside the LLN that may interfere with the interactions under consideration. To study the impact of such side traffic, we added another node that sends packets every 500 ms to the border router (Figure 3-21). The result is depicted in Figure 3-22. As expected, due to the additional packets at the border router, the latency has shown some increase.

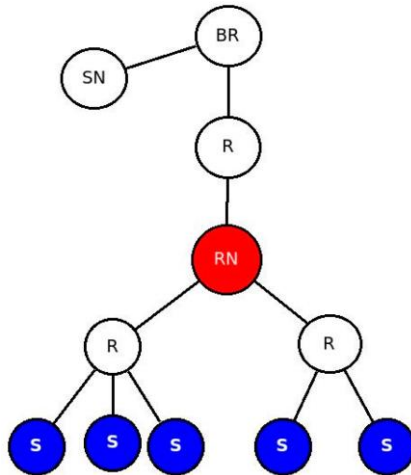


Figure 3-21: Network Topology including a Node Generating Side Traffic.

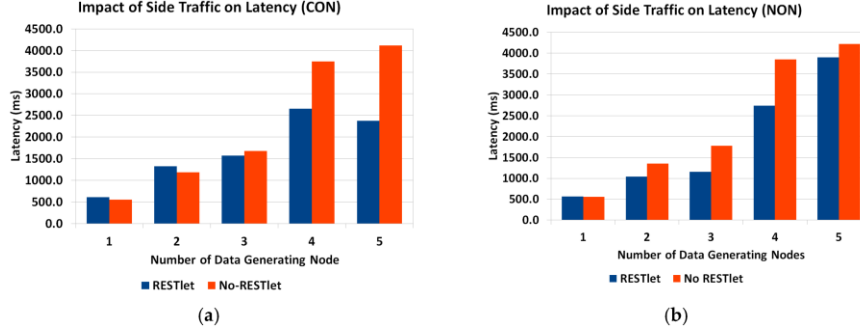


Figure 3-22: Impact of Side Traffic on Latency. (a) CONFirmable; (b) NON-CONFirmable transaction

D. Impact of Difference of Data Arrival Time

All the above tests showed that, the higher the number of packets that are being generated and transmitted inside the constrained network, the performance of both solutions, especially the NO-RESTlet solution, suffers. We run additional tests to observe the impact of the data arrival time difference on the latency by inserting an artificial gap in the data generation at the sensor nodes. All data generation nodes are made to generate data randomly between 0 ms and a maximum interval (this represents real world cases where multiple sensors observe the same physical phenomenon almost simultaneously). We used 500, 1000, 1500 and 2000 ms as maximum interval. The topology used is the same as the first test (Figure 3-18). We also run the experiment to observe the impact for number of data generation nodes. As can be seen in Figure 3-23, when there is no data generation gap (0 ms gap), the NO-RESTlet solution has much higher latency in most cases. The frequent arrival of packets at the border router creates congestion at the node. Due to the limited queue size of the constrained nodes, some packets will be dropped requiring retransmissions. This is the reason for the significantly high latency at 0 ms gap. When we look at the general trend in all graphs, the latency for both cases reduces until 1000ms data generation gap and starts rising slightly after that. The reason is simple. The introduction of artificial delays at the sensor node results in an additional delay in the end-to-end transmission. This means, the performance gain obtained by separating the arrival times will be countered by the artificial delay and as a result the overall latency starts to rise.

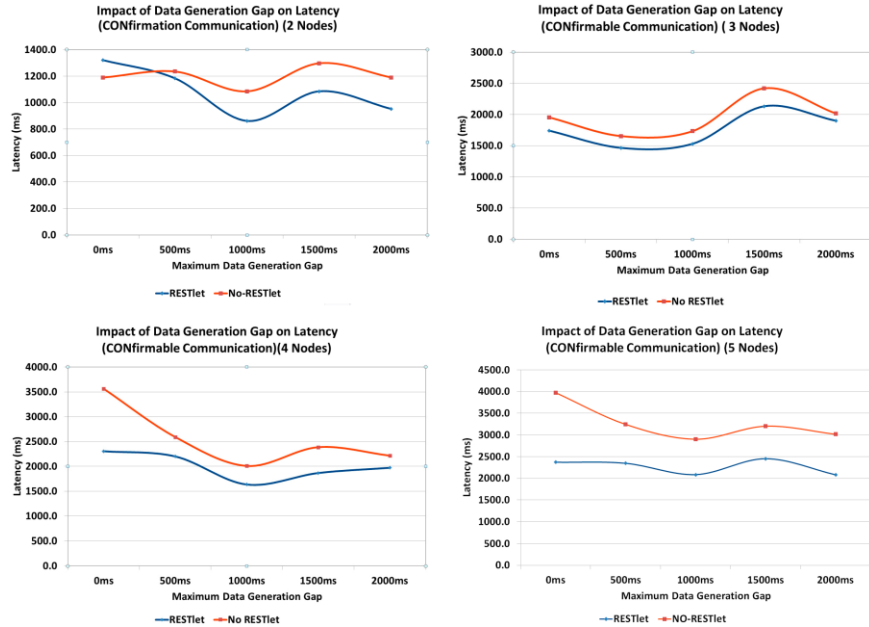


Figure 3-23: Impact of Packet Arrival Time Gap on Latency

E. Impact of Noise

Under normal working conditions, sensor and actuator nodes suffer from interference from other sources. This might create loss of packets requiring retransmissions in case of CONFirmable transmissions which, in turn, leads to increased latency. NON-confirmable transactions also suffer from increased latency since every lost packet leads to processing to be delayed until the 5 s interval is reached. To study the impact of lossy networks on latency, we run tests by setting the Transmission/Reception (TX/RX) loss from 0% (no loss), 5% and 10% losses. The topology we used is given in Figure 3-24. As the figure shows, there are three data generating nodes that send packets every five seconds without any time gap between the data generations. We selected three nodes to avoid the impact of having too many or too few data nodes which might skew the result to either side. Too little data generating nodes may influence the result in favor of NO-RESTlet case while too many nodes favor the other. The test is done both for CONFirmable and NON-Confirmable transactions.

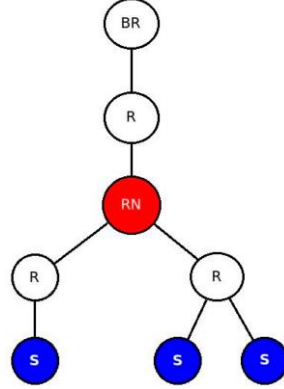


Figure 3-24: Network Topology for Noisy Networks

It is not a surprise that the results of the experiments in Figure 3-25 show higher overall latency for the NO-RESTlet case in both CON and NON communications. However, it is quite interesting to see that the difference between the NO-RESTlet and the RESTlet cases gets higher at higher TX/RX loss ratios. This indicates that, our solution is relatively more robust under lossy conditions for both CON and NON transactions.

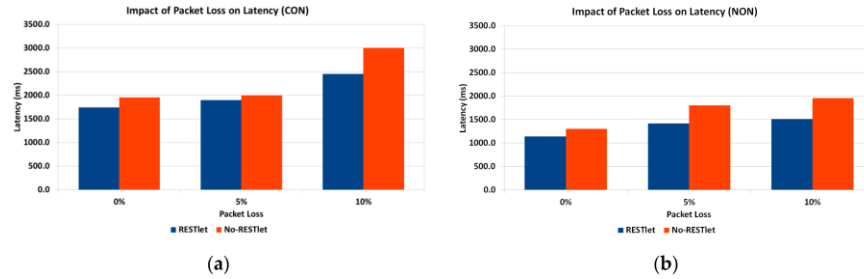


Figure 3-25: Impact of TX/RX Reception Ratio on Latency. (a) CONfirmable Communication; (b) NON-Confirmable Communication.

3.7 Conclusions and the Way Forward

In this paper, we presented two novel concepts that simplify sensor and actuator interactions and IoT application development by leveraging on CoAP as a protocol in combination with the Resource Observation extension. The binding concept effectively enables flexible direct interactions between sensors and actuators making gateway/cloud based solutions where intermediary devices accept input from sensors in order to trigger actuators redundant. The proposed solution, reduces the packet flow to the gateway and hence reduces latency and number of packets in the LLN compared to gateway or cloud based solutions. Through

experiments we showed that the overhead (e.g., memory footprint) introduced by the binding solution is not significant compared to the gateway/cloud based solutions. In fact, regarding many aspects such as communication delay and number of packets, the binding solution outperforms traditional solutions. We also showed that this flexibility can be achieved by only making minor changes to the CoAP protocol and the observe extension. The other novel concept, RESTlets, builds upon this binding concept. RESTlets are IoT application building blocks with data and control inputs, processing logic and data output. We showed that by using RESTlets as IoT application building blocks, we can do in-network processing and aggregation in order to reduce the number of packets that traverse the whole LLN to the edge of the network and/or to the cloud which otherwise would lead to higher latency. We also showed that by interconnecting the data inputs and outputs of RESTlets to sensor outputs, actuator inputs or other RESTlets, we can build a complete IoT application within the LLN. Since the RESTlet approach allows distributed deployment of the processing logic at different nodes, there will not be too many resource hungry processes on one single node. It also gives greater flexibility in developing IoT applications by placing simple processing functionality inside the LLN and more complex one at the gateway or in the cloud. We ran several experiments in order to evaluate the performance of our solution by comparing it to traditional gateway-based or cloud solutions by using a different number of data generating nodes, data generating gap and TX/RX ratio. In all cases, our solution is capable of outperforming traditional solutions in terms of latency. Interestingly, the RESTlet solution provides a very good opportunity to use visual programming techniques to reduce the IoT application development to a set of drag-and-drop or point-and-click activities. We do realize that this solution can be optimized further. One possible optimization could be achieved by looking at cross-layer processing activities. This is one of the potential areas of work in the future. In this paper, we stored the RESTlet code in the nodes at compile time which makes it inefficient in case that node is not selected to host that particular RESTlet. A more optimized solution would consist of the dynamic deployment of selected RESTlets at run-time. This is another area for future work. Optimal placement of RESTlet nodes in the network is also another future research topic. From the experiments we conducted, we found out that whenever the RESTlet is closer to the data generating nodes, the RESTlet solution performs better. In the future, we will come up with mathematical models which will lead to optimal placement of the RESTlet nodes.

Acknowledgments:

The research leading to these results has received funding from VLIR-UOS as a Ph.D. Scholarship to Girum K. Teklemariam through the Inter University Collaboration (IUC) Program at Jimma University, Ethiopia. Author Contributions: Girum K. Teklemariam wrote this paper as part of a Ph.D. thesis

under the supervision of Jeroen Hoebeke, Ingrid Moerman, and Piet Demeester. Floris Van den Abeele put forward several ideas and comments while defining the binding solution. Conflicts of Interest: The authors declare no conflict of interest.

References

- [3.1] Montenegro, G.; Kushalnagar, N.; Hui, J.; Culler, D. RFC4944—Transmission of IPv6 Packets Over IEEE 802.15.4 Networks. Available online: <http://tools.ietf.org/html/rfc4944> (accessed on 2 May 2016).
- [3.2] Brandt, A.; Hui, J.; Kelsey, R.; Levis, P.; Pister, K.; Struik, R.; Vasseur, J.P.; Alexander, R. RFC6550—RPL: Routing Protocol for Low Power and Lossy Networks. Available online: <http://tools.ietf.org/html/rfc6550> (accessed on 13 June 2016).
- [3.3] Shelby, Z. Embedded Web Services. *IEEE Wirel. Commun.* 2010, 17, 52–57.
- [3.4] Shelby, Z. RFC 7252 – The Constrained Application Protocol (CoAP). Available online: <https://datatracker.ietf.org/doc/rfc7252/> (accessed on 13 May 2016).
- [3.5] Hartke, K. RFC 7641: Observing Resources in the Constrained Application Protocol (CoAP). September 2015. Available online: <http://datatracker.ietf.org/doc/rfc7641> (accessed on 13 May 2016).
- [3.6] Teklemariam, G.K.; Hoebeke, J.; Moerman, I.; Demeester, P. Facilitating the creation of IoT applications through conditional observations in CoAP. *EURASIP J. Wirel. Commun. Netw.* 2013, 177. [CrossRef]
- [3.7] Kovatsch, M. Demo Abstract: Human—CoAP Interaction with Copper. In *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2011)*, Barcelona, Spain, 27–29 June 2011.
- [3.8] Shelby, Z.; Vial, M.V. CoRE Interfaces (Draft-Shelby-Core-Interfaces-05) (Work in Progress). March 2013. Available online: <https://tools.ietf.org/html/draft-shelby-core-interfaces-05> (accessed on 14 May 2016).
- [3.9] ZigBee Alliance. Zigbee Specification; ZigBee Standards Organization: San Ramon, CA, USA, 2008.
- [3.10] Lee, Y.; Liu, H.-S.; Wei, M.-S.; Peng, C.-H. A Flexible Binding Mechanism for Zigbee Sensors. In *Proceedings of the 5th International Conference on, Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, Melbourne, Australia, 7–10 December 2009; pp. 273–278.

- [3.11] Pautasso, C.; Zimmermann, O.; Leymann, F. RESTful Web Services vs. ‘Big’ Web Services: Making the Right Architectural Decision. In Proceedings of the 17th International World Wide Web Conference (WWW 2008), Beijing, China, 21–25 April 2008.
- [3.12] Guinard, D.; Ion, I.; Mayer, S. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers’ Perspective. In *Mobile and Ubiquitous Systems: Computing, Networking, and Services*; Springer: Berlin, Germany, 2011; pp. 326–337.
- [3.13] Kovatsch, M.; Lanter, M.; Duquennoy, S. Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications. In Proceedings of the 3rd International Conference on the Internet of Things (IoT), Wuxi, China, 24–26 October 2012; pp. 135–142.
- [3.14] Alessandrelli, D.; Patracca, M.; Pagano, P. T-Res: Enabling Reconfigurable in-Network Processing in IoT-Based WSNs. In Proceedings of the IEEE International Conference on Distributed Computing in Sensor Systems, Cambridge, MA, USA, 20–23 May 2013; pp. 337–344.
- [3.15] Azzara, A.; Mottola, L. Virtual Resources for the Internet of Things. In Proceedings of the IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, Italy, 14–16 December 2015.
- [3.16] Hughes, D.; Thoelen, K.; Horré, W.; Matthys, N.; Del Cid, J.; Michiels, S.; Huygens, C.; Joosen, W. LooCI: A Loosely-coupled Component Infrastructure for Networked Embedded Systems. In Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia (MoMM2009), Kuala Lumpur, Malaysia, 14–16 December 2009.
- [3.17] Shelby, Z.; Chauvenet, C. The IPSO Application Framework (Draftipso-App-Framework-04). Available online: <http://www.ipso-alliance.org/wp-content/uploads/2016/01/draft-ipso-app-framework-04.pdf> (accessed on 1 August 2016).
- [3.18] Dunkels, A.; Gronvall, B.; Voigt, T. Contiki—A Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, Tampa, FL, USA, 16–18 November 2004; pp. 455–462.
- [3.19] Kovatsch, M.; Duquennoy, S.; Dunkels, A. A Low-Power CoAP for Contiki. In Proceedings of the 8th IEEE International Conference on Mobile Ad-Hoc and Sensor Systems (MASS 2011), Valencia, Spain, 17–22 October 2011; pp. 855–860.

4

Dynamic Deployment of RESTlets on Constrained Devices

In the previous chapter, we introduced RESTlets as IoT application building blocks that can be hosted by any device including the constrained devices. The main limitation of our initial design of this concept for constrained devices, was that RESTlets had to be defined first on the device hosting them, after which they can be instantiated many times. Such a static definition of RESTlets on constrained devices implies that even if we do not need their functionality, they remain on the device and take up memory. This situation is not favorable for constrained objects. Therefore, in this chapter, we introduce a dynamic RESTlet deployment mechanism on constrained devices.

Girum Ketema Teklemariam, Floris Van den Abeele, Peter Ruckebusch, Ingrid Moerman, Piet Demeester, Jeroen Hoebeke. *Dynamic Deployment of RESTlets on Constrained Devices*. Submitted to International Journal of Distributed Sensor Networks May 2017.

Abstract: CoAP-based IoT applications can be developed in a distributed manner by using enablers such as RESTlets and Bindings. RESTlets accept inputs and produce outputs by defining processing logic of an IoT application. It can be hosted either on a constrained device or on non-constrained devices. Development of the IoT application is completed by interconnecting sensors, actuators and RESTlets by establishing a CoAP observe relationship among these components. This is what we call binding. Instead of deploying the RESTlets on constrained devices statically, we provided a mechanism of dynamic deployment of RESTlets on constrained devices, where they are deployed at run-time. Potential hosts are configured with a dynamic loader resource, which accepts dynamic modules sent through the CoAP Block-wise transfer method. Once all the blocks are accepted, the Contiki Dynamic Loader loads them in memory. The mechanism adds significant flexibility to development of distributed IoT applications.

4.1 Introduction

The term Internet of Things (IoT) has become a catch-phrase in just less than two decades of its first use by Kevin Ashton in 1999 [4.1]. There is no single definition for IoT yet. As per the recommendations of ITU-T Y.2060, ITU-T defines IoT as an infrastructure that “enables advanced services by *interconnecting* (physical and virtual) things based on existing and evolving *interoperable* information and communication technologies [4.2]. [4.3] defines IoT as scenarios where connectivity and processing capacity is extended to smart objects allowing them to generate and use data with minimal human intervention. These two definitions stress that there will be interconnections between the physical world (the “things”) and the virtual world using a plethora of technologies that needs to be interoperable. According to the definitions, autonomous operation is also key to the IoT concept where the “things” generate, exchange and use data with minimal human intervention giving more emphasis to Machine-to-Machine interactions in IoT systems. Such IoT systems are gaining more and more attention. Gartner Inc. identifies IoT as one of the top 10 strategic technology trends for the past 5 years [4.3]–[4.8]. There are also predictions that put the number of interconnected devices (“things”) to be more than 20 billion according to Gartner [4.9], and 50 billion according to Cisco and Ericsson [4.10], [4.11].

Due to the rapid growth of various technologies, different means of communications were introduced to realize the IoT vision. Most of these solutions have been developed keeping specific applications in mind since different application areas yield different constraints and requirements. This balkanization of efforts leads to vertical silos where “things” designed for one application (or designed by one manufacturer) cannot interact with applications or “things” coming from a different vendor. This limitation is a big challenge that needs to be

overcome in order to fully realize the IoT Vision where every “thing” is interconnected.

We may look at the problem from either an IoT application development framework perspective or a technology choice perspective. Absence of standardized development platforms force vendors to use proprietary solutions that are based on custom designed protocol stacks and communication modalities. Most of the “things” in the IoT paradigm are devices that have constraints in terms of memory capacity, processing and communication power. Due to this reason, vendors are tempted to come up with their own proprietary communication protocols among these devices leading to disjoint networks that cannot directly communicate with each other. As a result, vendors are forced to use (proprietary) gateways to interconnect their own set of devices with the Internet. This approach poses a serious interoperability issue if there are devices from multiple vendors in the same network. As a solution to this problem, IETF has been active in providing standardized communication protocols that are suitable for the constrained devices. 6LoWPAN [4.12], RPL [4.13], and CoAP [4.14] are some of the standards proposed by IETF to address these issues. In general, such standardization initiatives and efforts lead to the realization of IoT applications that can span across application areas and across different platforms.

To avoid the vertical silos and ensure fully connected IoT devices and applications, it is vital to use standardized protocols to build IoT applications. In this paper, we present a dynamic and CoAP-based distributed IoT application development and configuration model that can be deployed in the constrained network or low-power and lossy network (LLN), at the Gateway and/or in the cloud. It essentially breaks down IoT data processing into chains of processing blocks that can be distributed over all levels of an IoT system. At the heart of this model lies the concept of RESTlets, application building blocks that take input and produce outputs after performing some processing tasks. They also contain control parameters that can be used to configure the RESTlets dynamically at runtime. The RESTlets can be assigned to receive inputs from sensors (and other RESTlets) and send out their outputs to other RESTlets and/or devices (e.g. Actuators) using direct bindings. All interactions are realized by using the standardized CoAP protocol in order to achieve interoperability with limited overhead. A major challenge, however, is the dynamic deployment of such RESTlets in order to alter the way how data is being generated and processed down to the level of the most constrained devices. Assuming that application needs may change over time, it must be possible to alter the application level behavior of constrained devices during their lifetime. Our contribution is a system that dynamically deploys RESTlets at runtime offering significant flexibility for the deployment and configuration of RESTlet-based IoT applications. To validate our solution, we demonstrate how the concept of

Conditional Observation can be implemented using RESTlets which would otherwise require protocol modification.

The remainder of the paper is organized as follows. Related work in IoT application deployment and configuration options are discussed in the next section followed by a section that briefly describes RESTlets and Bindings, which are enablers for the distributed IoT application development. How conditional observation can be implemented using RESTlets is also discussed in this section. The fourth section describes RESTlet based IoT application development architecture and dynamic deployment of RESTlets in greater detail and explains how this concept can be used to implement conditional observation. The fifth section discusses results of functional and performance evaluations. The last section gives conclusion and indicates the way forward.

4.2 Related Work

As mentioned earlier, “things” in IoT applications are interconnected and should have the capacity to generate, exchange and consume data without (or with minimal) human intervention making Machine-to-Machine (M2M) communication a crucial component of IoT applications. Embedded Web services are considered to be an excellent mechanism for M2M communication [4.15]. They are tailored to constrained devices as they have very small communication overhead. One such a protocol is the Constrained Application Protocol (CoAP) [4.14]. CoAP is an open standard proposed by the IETF Constrained RESTful Environment (CoRE) working group. It is similar to HTTP in that it enables a RESTful approach where nodes expose their data as resource representations that can be accessed by GET, POST, PUT and DELETE methods [4.16]. Unlike HTTP, it uses UDP at the transport layer to avoid the overhead introduced by TCP. If reliability is required, packets can be transmitted as confirmable messages, which should be acknowledged by the receiver. In many IoT applications, data is generated by some nodes and is sent out to other nodes or to a central location. Resource observation is an important extension of CoAP that lets clients register their interest in resource state changes. As a result, the server will send notifications to the registered clients whenever there is a state change of that resource. Conditional observation [4.17], [4.18] further optimizes the observation by including notification criteria that tell the server to only send notifications when the criteria is met. This way, the number of packets in the network is significantly reduced especially in environments where small and insignificant changes in resource state are frequent.

Among the plethora of IoT applications that make use of web-services, a number of them are cloud-centric where the responsibility of the constrained devices is

limited to sending sensed data and/or affecting their environment [4.19]. In such systems, the entire intelligence of the applications resides in the cloud. In many cases, the results of the computations performed in the cloud are sent back to the constrained network. An alternative approach is performing some processing at the edge [4.20] or in the constrained network [4.21]–[4.24].

One of the challenges for using in-network processing for IoT applications is related to the flexible deployment and configuration of the IoT components that reside in the constrained network. [4.25] proposes a mobile phone based system which collects data from a variety of sensors and communicates the information to the application component in the cloud. This method relieves the communication burden of the nodes and also provides the possibility for the mobile phone to do prior filtering, if needed, before communication. This is considerably different from our work which focuses on dynamically deploying the IoT application development components anywhere in the network so that processing can take place in the constrained network, at the gateway or in the cloud. Another work on dynamic deployment is [4.26] which proposes a framework for IoT application development by delegating processing of some of the business logic to the edge of the constrained network. The framework targets the constrained network gateways and managing them. The framework provides the ability to deploy application components dynamically at several gateways and systematically manage them so that a large number of gateways, and hence a large number of constrained devices, are easily managed. Our work, however, allows the constrained nodes to do simple processing tasks.

4.3 RESTlets and Bindings

As mentioned before, in IoT applications, data processing can be done in the LLN, at the gateway or in the cloud. In this section, we will briefly explain how RESTlets are used to process data in the network and how they can be interconnected with each other and with other devices to build distributed IoT applications. Detailed description and evaluation of RESTlets and Bindings can be obtained in [4.27].

4.3.1 RESTlets

RESTlets are IoT application building blocks that accept data inputs and produce output after processing. They also have control parameters that can be used to dynamically configure the RESTlets (Figure 4-1). The inputs could be sensor outputs or outputs of other RESTlets while the outputs can be redirected, as an input, to another RESTlet, an actuator or a component of the IoT application at the gateway or in the cloud. A RESTlet can be placed on a node in the LLN or at the gateway or in the cloud. The processing logic actually defines the type of RESTlet.

On constrained devices we may have simple processing tasks such as averaging or thresholding while more capable nodes may perform complex tasks such as sending SMS or feature detection based on sensor data.

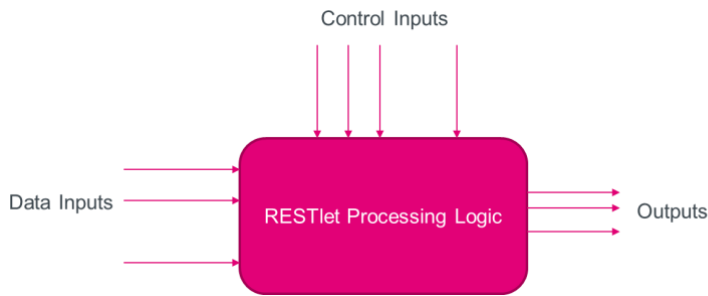


Figure 4-1 RESTlet

Using CoAP, we represent data inputs, control parameters and output of RESTlets as resources that can be manipulated and accessed using GET, POST, PUT and DELETE methods. In a very generic setup, in order to develop IoT applications using RESTlets, we first need to break down the problem that the application needs to address into smaller components and create RESTlets for each component. Next, we select nodes that will host the different RESTlets and deploy the RESTlets either dynamically or statically. Static deployment assumes the node possesses the knowledge on available RESTlet types as well as their implementation, allowing their on-demand instantiation. The selected nodes could be constrained devices, the LLN gateway or a component in the cloud. Once the RESTlets are hosted on devices, we can instantiate the RESTlets as required in order to make their functionalities available via CoAP resources. Finally, we interconnect the different components of the application (e.g. sensors, actuators, and other RESTlets) with each other to finalize the application development.

4.3.2 Bindings

Individual RESTlets are not usable unless the input and output resources are connected to some data sources and sinks. Sensors and actuators are very common data sources and sinks, respectively. Outputs of RESTlets can also be used as inputs for other RESTlets. To achieve such interconnection of components in a RESTful manner, we use bindings. Bindings are resource observation relationships between a client and server established by a third-party device that is not involved in subsequent interactions once the relationship is established. This is different from the normal CoAP observe mechanism [4.28] where clients register their observation interest after which they will receive all notifications themselves. In addition, the mechanism enables the generation of specific CoAP

requests upon the occurrence of a resource state change, instead of simply sending a CoAP request containing the new state (i.e. a notification).

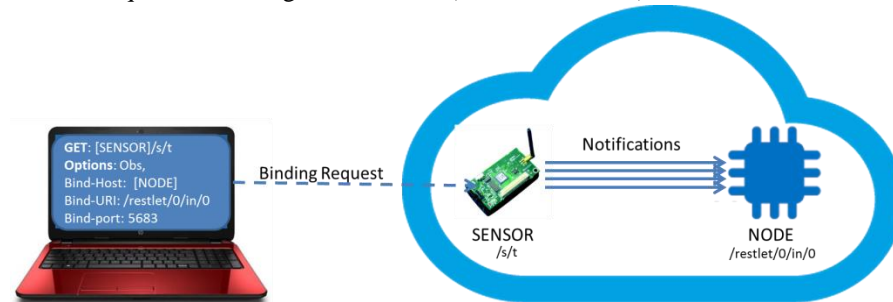


Figure 4-2: Binding Relationship Establishment

In order to create such flexible direct bindings, we must send a GET request to the node that hosts the observed resource, which contains the observe option as well as all details about who should be notified and how. This is done by adding a set of additional CoAP binding options (IPv6 address, port number, resource URI path and optional payload). Figure 4-2 illustrates how a binding relationship is established and how notifications are communicated. The sensor has an observable resource identified by `/s/t` and we would like this SENSOR node to notify the first input resource of the first RESTlet (represented by `/restlet/0/in/0`) hosted on the node NODE. To achieve this, we send a GET request that includes the CoAP observe option along with the binding options to the node [SENSOR] so that the observer is registered. Once the observer is registered, the SENSOR sends every resource state change directly to the `/restlet/0/in/0` resource on the node NODE. The binding request may be sent from any device connected to the Internet only once. All subsequent notifications do not involve the binder. The binding relationship can also be between a sensor and an actuator (e.g. a wireless light switch and a wireless light bulb.) A detailed description of the implementation and evaluation of bindings is given in [4.27].

4.3.3 Example 1 – Sample IoT Application

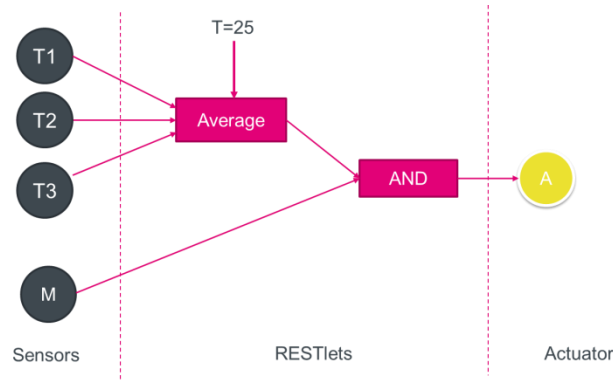


Figure 4-3: RESTlet-based IoT Application

To give a concrete view of the whole process, we consider the example of a simple home automation application that turns on/off the AC of an occupied room based on the average temperature of the room. Temperature values are obtained from three sensor nodes placed in the room and occupancy information is obtained from a motion sensor. Figure 4-3 shows a block diagram that shows how this can be achieved using RESTlets. For this simple IoT application, we need two RESTlets. The first RESTlet that performs the averaging has 3 inputs (each associated with a temperature sensor), an output and a control parameter (that indicates the cut-off point). The second RESTlet, AND (having 2 inputs and 1 output) performs a logical AND operation on the output of the AVERAGE RESTlet and the value obtained from the motion sensor. The output of this second RESTlet is connected to the actuator. Assuming that the first temperature sensor (T1) hosts the AVERAGE RESTlet and the motion sensor (M) hosts the AND RESTlet, the following sequence of CoAP requests will complete programming of the IoT application.

1. POST [T1]/r;		Payload: <RN=AVG;I=3;O=1;C=1>
2. POST [M]/r;		Payload: <RN=AND;I=2;O=1;C=0>
3. PUT [T1]/r/0/con/0;		Payload:<25>
4. GET [T1]/s/t;obs	=>	[S1]/r/0/in/0
5. GET [T2]/s/t;obs	=>	[S1]/r/0/in/1
6. GET [T3]/s/t;obs	=>	[S1]/r/0/in/2
7. GET [M]/s/m;obs	=>	[M]/r/0/in/0
8. GET [T1]/r/out/0;obs	=>	[M]/r/0/in/1
9. GET [M]/r/0/out/0;obs	=>	[A]/a/t

Figure 4-4: Statements to Create RESTlet-based IoT Application

Figure 4-4 shows list of statements needed for the creation of the RESTlet-based IoT application. The first two statements create the RESTlets on the nodes T1 and M while the third statement initializes the first control parameter of the AVG RESTlet to 25. Statements 4 through 7 bind the sensor outputs to the respective inputs of the RESTlets. Statement 8 associates the output of the AVG RESTlet (on node T1) to the first input of the AND RESTlet (on node M) while the last statement binds the output of the AND RESTlet with a resource on the actuator (A).

4.3.4 Example 2 – RESTlets and Conditional Observe

Conditional observation [4.17], [4.18] extends the CoAP observe option by giving server-side filtering possibility. This feature is realized through the definition of a new *Condition* option. Clients include this newly introduced *Condition* option along with the Observe option to specify their observation interest and criteria of notification. The condition option (Figure 4-5) could be between 1 to 5 bytes in size. The five most significant bits of the most significant byte, labeled CT in the figure, are used to indicate the condition type, allowing 32 possible condition types. The RF bit is a reliability flag indicating whether the response should be sent as a confirmable or non-confirmable message. The last two bits indicate the data type of the value. Currently, there have been 3 types identified, namely, integer, float and duration. The following bytes, if available, are the actual filtering conditions or threshold values.



Figure 4-5: Conditional Observe Option Format

All interested observers register their interest along with the filtering criteria by sending a CoAP observe request to the server by setting the appropriate values for the Condition Option. Once the registration process is complete, the server checks if the condition is met before notifying the observer whenever there are resource state changes.

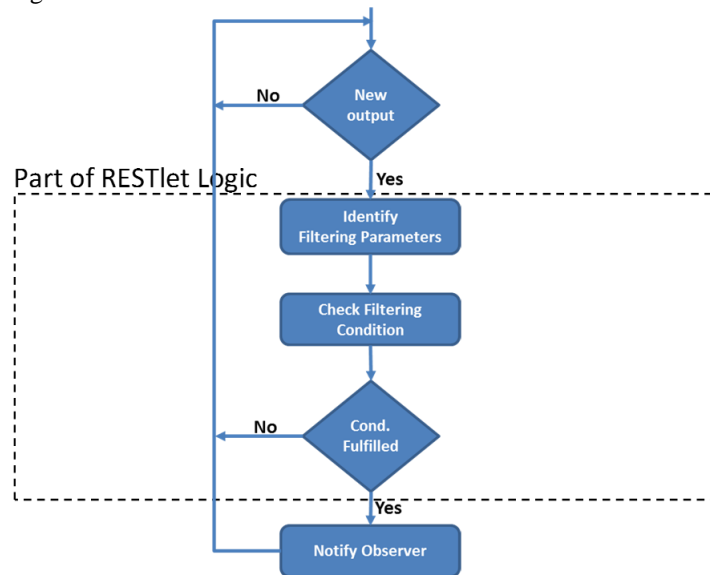


Figure 4-6: Conditional Observation Processing Logic Flow-chart

Conditional observe is beneficial in order to reduce the number of messages exchanged in the constrained network. However, it requires an extension of the CoAP protocol, i.e. the implementation of the *Condition* option on all involved devices (clients and servers). As an alternative, the RESTlet concept can be used to emulate the conditional observe functionality by defining a RESTlet that changes its output only when a certain condition is met. The RESTlet will have at least 1 input, a control parameter and at least 1 output. If multiple sensors exist on a single node, we can simply increase the number of inputs, controls and outputs, enabling the behavior for multiple sensors. Moreover, if clients are interested in different condition values of the same sensor, we have to use a different control parameter and output set. The input of the RESTlet will be associated with the sensor output, i.e. the measured value. The control parameter holds the information that is normally stored in the Condition Option. Changing the value

of the control parameter at runtime (using PUT requests), effectively changes the RESTlet behavior by changing any of the condition option components. The processing logic detects the condition type and performs the appropriate filtering on the data to decide whether the new output needs to be communicated to observers or not as shown in the flow chart (Figure 4-6). As a result, similar functionality is achieved without requiring any extension to the CoAP protocol. This is a clear example of how the RESTlet concept can be used to enrich the functionality of a node, after deployment and depending on the needs of an application.

4.4 Architecture and Dynamic Deployment of RESTlets

In IoT systems that use web services (e.g. CoAP), data is generated by constrained nodes to be consumed by other devices. The data may have to be processed before it is used. In a generic system, data processing can be done by different devices based on the device capacity and the requirements of the application (Figure 4-7). As shown in the figure, we may see different approaches in IoT application development. In the first approach, data is processed inside the constrained network by selected constrained nodes. This may be achieved by performing simple processing or aggregation tasks by the constrained nodes and possibly consuming the resulting data within the constrained network. A completely opposite approach is a cloud-centric [4.29] approach where every piece of data is sent to the cloud for processing and the result is sent back to the constrained network if required. Processing can also be done at an intermediate location such as a gateway. Conditional observation mentioned in the previous paragraph is an example of In-Network processing while Actinium [4.30] is a cloud centric solution.

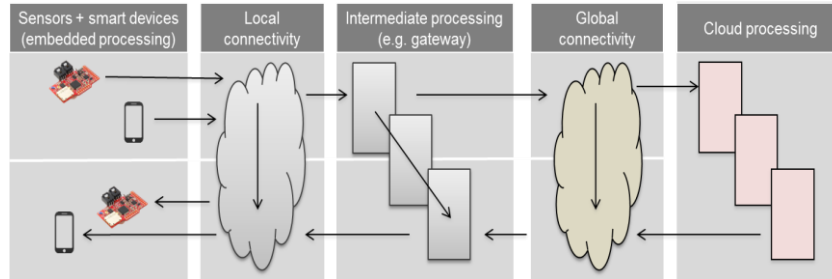


Figure 4-7: Generic IoT System

As discussed above, the RESTlet concept can be used to realize such generic IoT systems. One possible architecture to realize such a generic IoT system using RESTlets is given in Figure 4-8. The generic RESTlet architecture shown in the figure illustrates different constituent parts of a RESTlet-based system as it resides on a single node.

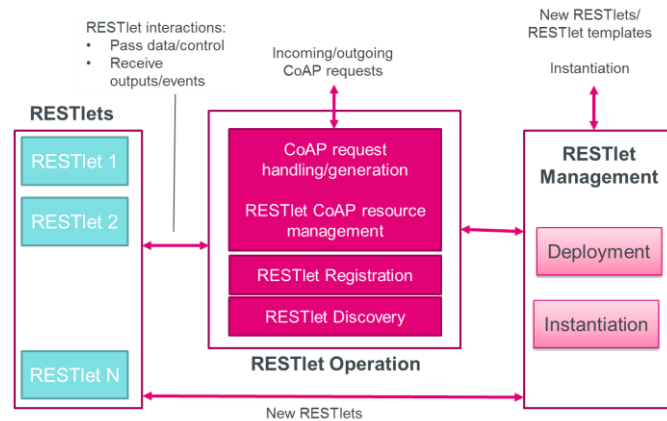


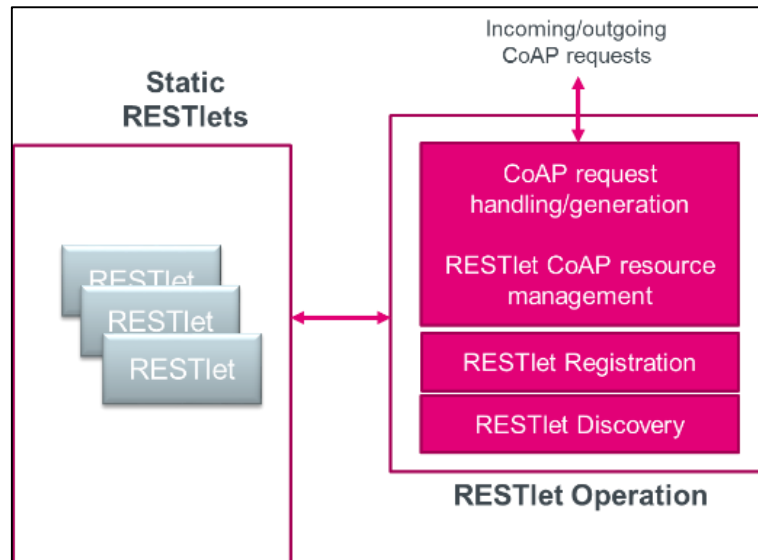
Figure 4-8: Generic RESTlet Architecture

Depending on its capacity, a node may have multiple RESTlets, each RESTlet having its own processing logic and associated interfaces as CoAP resources. We may have different options for deployment (static vs. dynamic) and instantiation of the RESTlets. The RESTlet management module takes care of these tasks. If the RESTlet template already exists on the node, the RESTlet management module just sends instantiation message to the operation module so that the required resources are created. Alternatively, if new RESTlets are required, the management module dynamically deploys the RESTlet and informs the details of the new RESTlet to the operation module. If necessary, the dynamically deployed RESTlet can be instantiated immediately. The RESTlet operation module provides interfaces to use CoAP methods to interact with RESTlet resources. The RESTlet

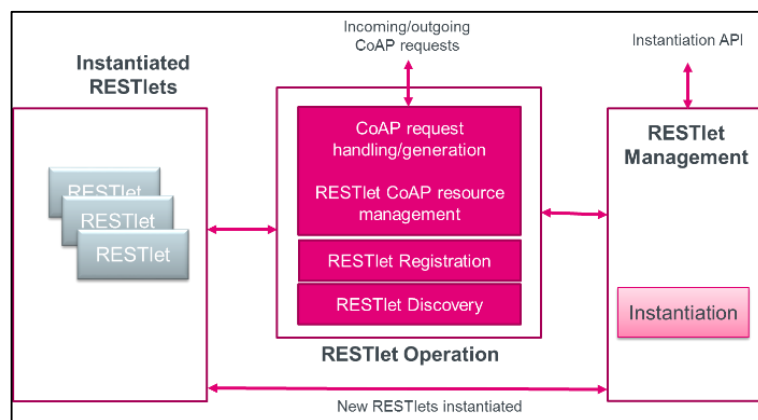
CoAP resource management part of the RESTlet operation module performs tasks such as creating of RESTlet CoAP resources when a RESTlet is instantiated while the CoAP request/resource handler responds to requests. Whenever a request arrives, the data and control parameters are sent to the RESTlet which returns the output after processing the data. The handler then sends out the processed output to the client. A list of instantiated RESTlets should also be maintained to keep track of all available RESTlets and their instances and make them ready for discovery. The RESTlet Registration and Discovery components of the RESTlet operation modules accomplish this task.

In such generic systems, a CoAP resource, named */restlet*, may be used as both a management interface for instantiating new RESTlets as well as an entry point to the RESTlets that are already defined (or deployed) on a node. Sending a CoAP POST request to this resource by specifying the name of the RESTlet, the number of inputs, outputs and control parameters as a payload instantiates the RESTlet. This implies that a new CoAP resource is created and activated for each input, output and control parameter. The resulting resources may be accessed in a way that uniquely identifies the specific RESTlet instance and its interfaces. For instance, the resource */restlet/0/in/1* may refer to the second input resource of the first RESTlet and */restlet/1/con/2* may refer to the third control parameter of the second RESTlet. Further lookups may be used to reveal the names of the RESTlets (which is not actually required for the IoT application development). A RESTlet may be instantiated multiple times depending on the application. For instance, the AVERAGE RESTlet may be defined twice on a node to compute the averages of two different sets of sensor readings. Once a RESTlet has been instantiated, further interactions can be made to the resources by sending GET and PUT requests to specific resources. For example, a PUT request sent to the */restlet/0/in/1* will update the second input resource of the first RESTlet while a GET request to */restlet/0/out/0* results in a response with the current representation of the first output of the first RESTlet. Changes in input may result in calling the processing logic which in turn may result in new output(s). Outputs are made observable so that changes are immediately communicated to observers. PUT requests sent to the control parameters may be used to configure the RESTlets by changing some RESTlet parameters at runtime. DELETE may be used to remove a RESTlet instance along with its resources or delete a specific resource of an instantiated RESTlet (e.g., an input resource of a RESTlet).

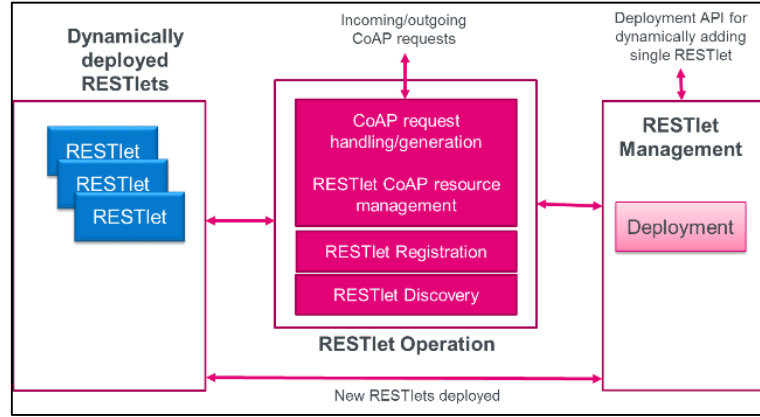
However, due to the fact that RESTlets can be placed on various devices with different capabilities and lack of a priori knowledge of RESTlets required for an application, a single implementation option of this architecture leads to inefficient IoT applications. High level implementation possibilities of such a RESTlet architecture are given in Figure 4-9.



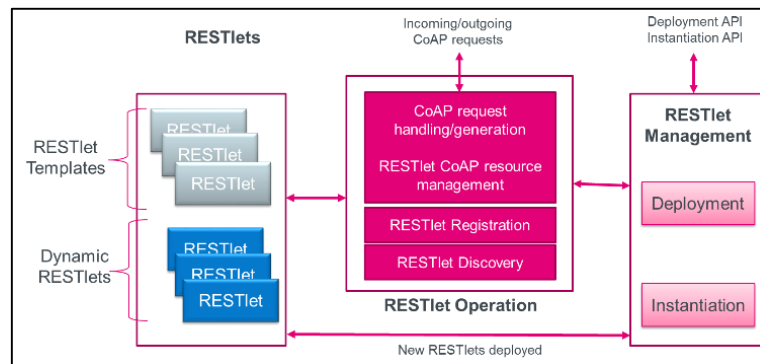
(a)



(b)



(c)



(d)

Figure 4-9: Implementation Options of Generic RESTlet Architecture. a) Static. b) Template Based. c) Dynamic. d) Hybrid

Figure 4-9a depicts a static implementation where all RESTlets and RESTlet instances the IoT application needs are known and are hard-coded. In this approach, all RESTlets are compiled into the application and will be instantiated when the application is loaded. Therefore, there is no need for RESTlet management module since deployment and instantiation is already done. A relatively flexible implementation option is given in Figure 4-9b. In such template based implementations, the templates of the RESTlets that might be needed are already loaded into memory along with the application. But, instantiation is done only if there is a need for an instance of a specific RESTlet template. Multiple instantiations of the same RESTlet template is also possible. For such solutions, RESTlet instantiation API must be provided to interact with the RESTlet management module to enable instantiation at run-time. In cases where a priori knowledge of required RESTlets doesn't exist before hand, dynamic deployment

is a better solution. The application will be loaded without RESTlets but APIs for deployment of RESTlets will be provided (Figure 4-9c). When the need arises, a RESTlet will be dynamically deployed and instantiated immediately. This is a very flexible option that can be implemented on a very constrained device too. The last implementation option, shown in Figure 4-9d, is a hybrid system which supports every possible method of using RESTlets. Pre-defined RESTlet templates can be instantiated at loading time or on-demand. In addition, new RESTlets can be created dynamically on the fly. The dynamic RESTlets can also be instantiated immediately after loading or later-on. To provide this flexibility, both deployment and instantiation API's needs to be provided.

The Need for Dynamic Deployment

The previous subsection described a generic architecture for the RESTlet concept that can be implemented on a constrained or non-constrained device. The static and template based mechanisms require nodes to already define the RESTlets statically as part of their firmware or code base. However, due to the resource scarcity of constrained devices, a priori definition and subsequent instantiation of multiple RESTlets may be too heavy for nodes with significantly small memory. In addition, static deployment and instantiation is not flexible as it may require flashing of the whole firmware of a node just to add a new RESTlet to it. Therefore, a better option for constrained devices is to dynamically deploy and instantiate RESTlets on selected nodes at runtime.

Dynamic deployment is a better alternative for non-constrained devices too, but for a whole different reason. Due to the distributed nature of RESTlet-based IoT applications, we do not have prior information about which RESTlets are best hosted on which device. It is not also feasible and efficient to define all possible RESTlets on all involved non-constrained devices beforehand. This also calls for dynamic deployment options that allow dynamic creation and removal of RESTlets.

4.5 Implementation and Evaluation

4.5.1 Implementation of Dynamic RESTlets on Constrained Devices

As this work is a continuation of earlier work on bindings and RESTlets, we used similar tools to implement dynamic RESTlets and evaluate the performance. For constrained devices, we based our experiments on Contiki 2.7 [4.31]. The μ IP implementation of Contiki along with 6LoWPAN and RPL are used for communication between nodes. Erbium [4.32], a CoAP implementation for

Contiki, is also used and extended to support bindings and dynamic deployment of RESTlets. For non-constrained devices, we used CoAP++, our in-house CoAP framework.

As described in detail in [4.32], Erbiun needs to be modified to support direct flexible bindings and RESTlets. To support bindings, four new CoAP options have been introduced, namely `BIND_URI_HOST`, `BIND_URI_PORT`, `BIND_URI_PATH` and `BIND_PAYLOAD` with option numbers 42, 46, 50 and 54. These options are used in a request in combination with the CoAP observe option. The first three options are used to uniquely specify the resource that needs to be triggered upon the state change of the resource targeted in the observe request. The CoAP PUT method is used in the notification packets. The optional `BIND_PAYLOAD` option contains the payload that needs to be sent. In its absence, the new state representation of the observed resource that triggered the notification is communicated.

The RESTlets on constrained and non-constrained devices are implemented following different implementation options as explained in section 4. Figure 4-10 shows implementation of dynamic RESTlets on constrained devices.

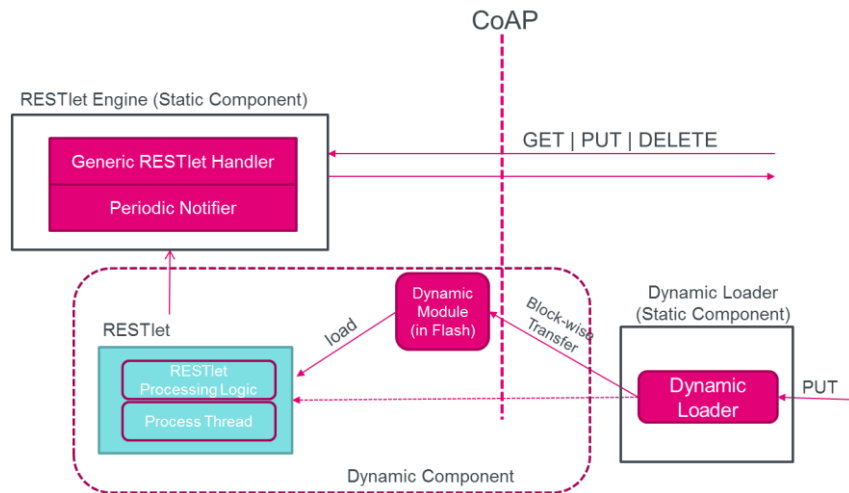


Figure 4-10: Implementation of Dynamic RESTlets on Constrained Devices

There are static components that are part of the device firmware and a dynamic component that is loaded at run-time. All RESTlet-ready nodes will have a simplified RESTlet engine which contains a generic handler function that facilitates CoAP interactions with the RESTlet resources. In addition, a function is defined to allow periodic notification of RESTlet output resource state changes. Unless a RESTlet is deployed and instantiated, the RESTlet engine will not be

activated to avoid wastage of CPU processing cycles. We also need a data structure that stores all relevant information about a RESTlet that is dynamically loaded. Table 4-1 shows the resulting structure. Finally, the dynamic loader is another component that is also available statically.

The dynamic deployment process starts when the user sends a CoAP PUT request to the Dynamic Loader (the */dloader* CoAP resource). The payload of the request is the binary file of the dynamic module. Due to its relatively large size, a bulk transfer mechanism, such as the CoAP block-wise transfer [4.33], must be employed to transfer the binary file. Upon successful transfer, the code resides in an external memory of the node until it is loaded into memory. The Contiki dynamic loader loads the dynamic module into memory (RAM/ROM) to be called by the RESTlet engine. During the loading process, all relevant RESTlet resources will be created and the RESTlet engine will be activated so that the RESTlet is instantiated and is ready for use. In our current implementation, we have only one RESTlet and one instance of that RESTlet on a node but this can be easily extended in order to support more RESTlets. Therefore, the RESTlet resources may be accessed using the name of the RESTlet and the type of component and the component number. For example, the first input of the AVG RESTlet is referenced using the resource */AVG/in/0*. Once the loading process has completed successfully, further bindings, notifications, resource state modifications and retrievals may commence using the available CoAP methods in the same manner as the static RESTlets discussed in [4.27].

Table 4-1: RESTlet Structure

Field Name	Possible Data Type	Description
name	Character array	Stores the name of the RESTlet. Currently, the name is the only field that describes the purpose of the RESTlet
number_of_input	Unsigned Integer	Number of input resources
input	Array of Values	Current Input values obtained from sensors or other RESTlet outputs
number_of_output	Unsigned Integer	Number of Output resources
output	Array of Values	Current Output Values obtained as a result of the processing logic
number_of_control	Unsigned Integer	Number of Control Parameters
control	Array of Values	Current configuration parameters
processing_logic	Function Pointer	The address of the processing logic

When we look more closely at the dynamic module, we see that all RESTlets have a similar structure. The first and most important component is the processing logic. Depending on their purpose, every RESTlet should define a function that takes input data and (optionally) control parameters to produce output(s). We also need

to transfer the details of the RESTlet to the RESTlet engine, which is part of the statically loaded firmware. This is achieved by defining the *restlet* structure as an external global variable in the dynamic module and then initialize the variables using RESTlet specific information. The RESTlet name, the number of inputs, outputs, control parameters and address of the function implementing the RESTlet processing logic are some of the values that have to be initialized. In Contiki-based systems, the best place to do such initializations is the main process thread which is the other component of the dynamic module. The main process thread is defined as an *autostart* process that will be automatically run after the dynamic loader has loaded the module. Optionally, the function(s) that update the inputs and control parameters may be defined here in the dynamic module. This requires adding their function pointer in the *restlet* struct. But we need to carefully weigh the trade-off between size of the dynamic module and memory footprint of the static firmware. If we keep these functions in the dynamic module, the size of the module will become too big, requiring more packets to transfer the RESTlet code, in turn leading to increased power consumption. On the contrary, if we keep it in the firmware, it takes more memory from the very beginning even if no RESTlet has been loaded. In all our tests, the input and control parameter updating functions are defined statically as part of the RESTlet engine.

Once the relevant components have been defined, the dynamic module can be compiled as a normal Contiki application using the Contiki make system. However, since this is a dynamic module, we need to strip off unnecessary symbols and keep only the relevant components. For instance, we do not need the entire communication stack (RDC drivers, MAC drivers, uIP, RPL, 6LoWPAN ...), sensors, and radio drivers for the dynamic module. The compiler removes all these and other irrelevant components and only keeps the components that we defined earlier in the dynamic module. In addition, since we use the *restlet* struct from the firmware to be used by the dynamic module, we must make sure that it is included in the symbol table of the firmware so that proper relocation of code takes place.

The compiled dynamic modules are deployed on selected nodes using CoAP block-wise transfer. We can send the file by assigning different block sizes for the CoAP block1 option. Since we need all the blocks to arrive for proper deployment, we use CONfirmable messages for the block-wise transfer. Contiki uses different mechanisms to load executable modules dynamically. In our tests, we used the Contiki *elfloader* [4.34]. The *elfloader* takes Executable and Linkable Format (ELF) files stored on external storage as input and loads them in memory (RAM or ROM) after performing all the required relocations. Furthermore, Contiki uses different Makefiles to set different compiler and loader flags to build the firmware and the dynamic modules. We made some modifications to one of the Makefiles

and the elfloader so that the dynamic modules are loaded properly. The Makefile that needs modification is the CPU specific Makefile which defines parameters specific to the CPU of the node. We are using Zolertia nodes [4.35] which use a MSP430 [4.36] microcontroller. Some parameters of the MSP430 Makefile put the code in the appropriate memory region so that they are relocated and loaded accordingly. The Contiki elfloader also expects specific code segments to be placed in specific regions. Therefore, the MSP430 Makefile and the elfloader should have a common name and location for the different components. Therefore, we made some changes to the Makefile and the elfloader.

The Makefile is modified to load the data and code segments of the firmware and the dynamic module anywhere in memory (in the lower or upper memory area of the Zolertia nodes) depending on the overall size of the firmware. However, to improve the memory usage of the firmware, each function and data are made to be placed in separate sections. This is not required for the dynamic module as it is small in size. Moreover, doing so for the dynamic module will increase the complexity of the elfloader. So, the Makefile is made to selectively set the appropriate flags for the dynamic module and the firmware (Figure 4-11).

```

TARGET_MEMORY_MODEL = medium
CFLAGS += -mmemory-model=$(TARGET_MEMORY_MODEL)
CFLAGS += -mcode-region=any
ifndef DYNAMIC_MODULE
    ${info INFO: Using Medium Memory for Static Firmware }
    CFLAGS += -ffunction-sections -fdata-sections
else
    ${info INFO: Using Medium Memory for Dynamic Module}
endif
LDFLAGS += -mmemory model=$(TARGET_MEMORY_MODEL)
LDFLAGS += -Wl,-gc-sections

```

Figure 4-11: Instructions from MSP430 Makefile

The elfloader also needs to be modified in order to handle the dynamic module. The default Contiki elfloader recognizes .text, .data, .bss, and .rodata sections and their corresponding relocation sections of the dynamic module and looks for them while loading. However, the MSP430 Makefile flags put the dynamic module's text and data segments (and their associated relocation segments) in .any.text and .any.data segments. This new set of segments should be included in the list of sections the elfloader searches in place of their original counterparts. These two modifications enable proper loading and initialization of the dynamic module.

4.5.2 Implementation of Conditional Observe using Dynamic RESTlets

The previous discussions show how RESTlets can be deployed dynamically on constrained devices and how they can be used to build IoT applications. Here, we will show how dynamic RESTlets can be used to implement conditional observation. The simplified block diagram is shown in Figure 4-12.

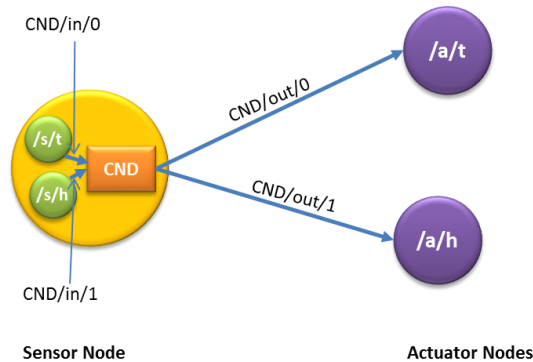


Figure 4-12: Block Diagram of Implementation of Conditional Observe Using Dynamic RESTlets

In order to implement conditional observations using RESTlets, we need to define one RESTlet per sensor node. Multiple input, output and control parameters can be defined for each sensor on the node. The figure shows two sensors, whose resources are represented by `/s/t` and `/s/h`, on a single node that performs the server-side filtering. The resources of the two sensors are associated with two inputs of the CND RESTlet. The outputs of the RESTlet is associated with two actuators. (It is also possible to associate them with the same actuator depending on the application and purpose of the actuator.) The processing logic of the RESTlet makes use of the control parameters (not shown in the diagram) that are defined for each set of input and output. The control parameters contain important information such as condition type, reliability flag, value type and condition value. The processing logic makes use of the corresponding control parameter on the input to determine the output. There are various ways to implement the processing logic and produce different types of output. For the sake of experimentation, we made the output to be 1 if the sensor output (which is the input to the RESTlet) should be communicated and 0, otherwise.

4.5.3 Functional Evaluation

4.5.3.a Compilation of the Dynamic Module

As explained before, the dynamic loading process starts with compiling the dynamic module with the appropriate flags. Figure 4-13 shows the screenshot of compilation process of the CND RESTlet. The Contiki make rule compiles the program as a Contiki application and strips off unnecessary components from the file.

```
$ make cnd.ce DYNAMIC=1
INFO: compiling with CoAP-13
using saved target 'z1'
INFO: Using Medium Memory for Dynamic Module
CC      cnd.c
msp430-strip --strip-unneeded -g -x cnd.ce
```

Figure 4-13: Compiling Dynamic RESTlet (CND)

Upon successful compilation, a dynamic module named `cnd.ce` will be created. This module can be uploaded by the Contiki `elfloader` later on.

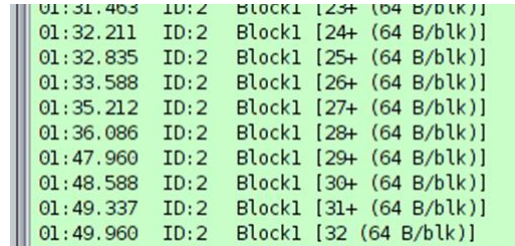
4.5.3.b Block-wise Transfer

CoAP++ allows transferring large binary files through CoAP Block-Wise transfer. We used this functionality to transfer the dynamic module to the host node. Table 4-2 summarizes the most important set of information required for the transfer.

Table 4-2: Information Required to Transfer the Dynamic Module

Field/Option	Value	Description
METHOD	PUT	
URI	coap://[aaaa::c30c:0:0:2]:5683/dload	The host node's IP address and the dynamic loader resource
MESSAGE_TYPE	CON	Message MUST be confirmable
BLOCK_SIZE_BLOCK_PAYLOAD	BLOCK_SIZE_64	Block1 size is 64 bytes
PAYLOAD_MEDIA_TYPE	COAP_MEDIA_TYPE_APPLICATION_OCTET_STREAM	Payload is binary file
PAYLOAD_FILE_NAME	base64(cnd.ce)	Base64 encoded name of the dynamic file.

CoAP++ creates packets containing the contents of the binary file as payload and starts transferring the file to the selected node. The receiving node stores the file in the Contiki file system on an external flash. The screenshot (Figure 4-14) shows the time of arrival of the block, including the Node ID, block 1 size and block number. All blocks except the last one are suffixed with a '+' indicating subsequent blocks are following. In this case, as soon as Block 32 has been received, transfer of the entire dynamic module is complete and loading may follow.



01:31.463	ID:2	Block1	[23+ (64 B/blk)]
01:32.211	ID:2	Block1	[24+ (64 B/blk)]
01:32.835	ID:2	Block1	[25+ (64 B/blk)]
01:33.588	ID:2	Block1	[26+ (64 B/blk)]
01:35.212	ID:2	Block1	[27+ (64 B/blk)]
01:36.086	ID:2	Block1	[28+ (64 B/blk)]
01:47.960	ID:2	Block1	[29+ (64 B/blk)]
01:48.588	ID:2	Block1	[30+ (64 B/blk)]
01:49.337	ID:2	Block1	[31+ (64 B/blk)]
01:49.960	ID:2	Block1	[32 (64 B/blk)]

Figure 4-14: Cooja Simulation Showing Transfer of the last Blocks

4.5.3.c Loading and Instantiation

During a CoAP block transfer, every packet shows the availability of more packets as part of the CoAP Block option. After the last block has been transferred, the loading process begins which performs relocation and copies the code and data into memory (RAM or ROM). The loading process culminates by calling the main process thread of the dynamic module which is defined as an automatically started process. The main process thread initializes the *restlet* structure which is defined to store details of the RESTlet being created. Finally, the RESTlet will be instantiated by creating resources. Once the instantiation is completed, the RESTlet may be used in the same way as it was defined statically as detailed in [4.27].

4.5.4 Performance Evaluation

We used Zolertia Z1 motes for performance evaluation. The motes have 8KB RAM and 92KB ROM. They also have 16MB external flash that can be used to store the dynamic module before loading. Since different RESTlets have different size, the type of RESTlet considered will have an impact on memory footprint and transfer time. For this reason, we used 3 RESTlet types. The first one is the AND RESTlet which performs the logical AND operation on all of its inputs to provide an output. The RESTlet may have any number of inputs but produces only one output with value either 0 or 1. The default number of inputs used for this test is 2. The second RESTlet defined was AVG that outputs the average value of the inputs supplied. Here the default number of inputs considered is 2 as well and the number of outputs is one. These two RESTlets do not use any control parameter. The last RESTlet defined is the CND RESTlet which is the implementation of conditional observation using RESTlets. Unlike the previous two RESTlets, which use all inputs to produce an output, this RESTlet uses one input source and one control parameter to produce an output. The input is assumed to be associated with a sensor on the same device. If multiple sensors exist on one node, we define additional input and associated control and output resources. The control

parameters store the encoded conditional observation information and have to be provided when establishing the binding.

We run several tests to measure energy consumption and transfer time for dynamic deployment of RESTlet modules using different radio duty cycling (RDC) options, namely ContikiMAC and NULLRDC. We also evaluate the impact of hop count on the transfer time and power consumption.

4.5.4.a Memory Footprint

One of the constraints of smart nodes is memory. According to RFC7228 [4.37] the most constrained devices, Class 0 devices, have less than 10KiB of RAM and less than 100KiB of ROM. The memory footprint of our implementation to support the dynamic deployment of RESTlets is given in Table 4-3. The first two rows of the table show the memory footprint of the firmware with dynamic modules (The Contiki Dynamic loader allows placement of dynamic modules in RAM or ROM). In both cases, the RAM requirement is less than 8KiB and the ROM requirement is less than 100KiB making the solution applicable to Class 0 devices such as the Zolertia Z1 motes. Of course, the solution uses more RAM and ROM as compared to its static counterpart (last row). However, the important issue here is not the actual amount of memory used. As long as the program code and data fits in a constrained device of Class 0, it is a viable solution for IoT application development.

Table 4-3: Firmware Size (Byte)

Firmware Type	Text	Data	BSS	Total
Firmware with Dynamic Module (in ROM)	57255	390	6374	64019
Firmware with Dynamic Module (in RAM)	56261	390	7138	63789
Firmware with Static RESTlet	49138	360	6184	55682

Closing up on the three architectural components of our RESTlet architecture (Table 4-4), we observe that the most significant increase in memory consumption is due to the dynamic deployer that contributes to more than 95% of the extra memory (Figure 4-15). Apart from this, the RESTlet engine and the RESTlets themselves do not contribute a lot to the extra memory. The main reason for the excessive memory usage of the deployer is the inclusion of the Contiki elfloader and file system modules. The former was added to enable dynamic deployment while the later was added to allow the node to store the dynamic module in external memory before loading.

Table 4-4: Sizes of RESTlet Architectural Components

Component	Dynamic Loading				Static Loading			
	TEXT	DATA	BSS	Total	TEXT	DATA	BSS	Total
RESTlet Engine	1513	28	393	1934	1103	28	392	1523
Deployer / Instantiator	7909	40	130	8074	110	0	20	130
RESTlets								
RESTlet AND	146	12	0	158	40	0	0	40
RESTlet AVG	192	12	0	204	86	0	0	86
RESTlet CND	496	12	12	520	398	0	12	410

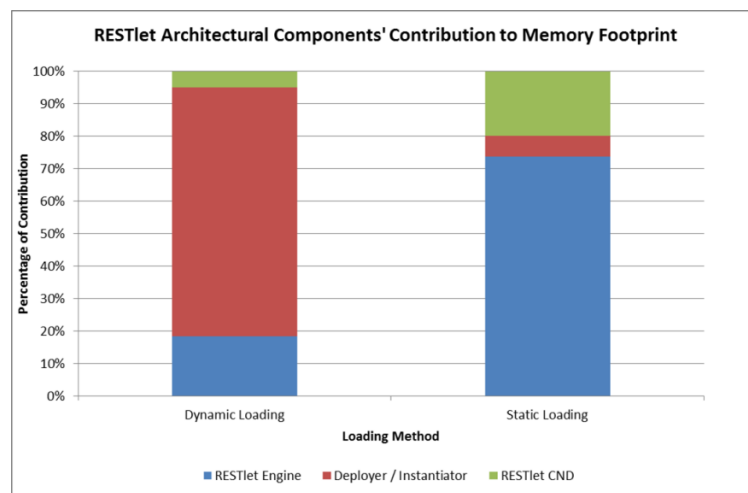


Figure 4-15: Contribution of RESTlet Architectural Components to the Overall Memory Footprint

4.5.4.b Number of Packets

The size of the dynamic modules on disk is much larger than their size in memory. In addition to the program code and data, the file on disk contains extra information such as section headers, symbol tables, string tables and relocation entries. The Contiki dynamic loader uses this extra information to properly relocate the data and code segments and load them in the appropriate memory location. The entire file on disk needs to be transferred to the node prior to loading into ROM or RAM of the node. Table 4-5 shows the size of the 3 RESTlets on disk and the number of packets required to transfer them to the constrained node.

Table 4-5: Number of Packets Required to Transfer RESTlets

RESTlet	File size	Number of PKTs (1 Hop)			Number of PKTs (2 Hops)			Number of PKTs (3 Hops)		
		Block 16	Block 32	Block 64	Block 16	Block 32	Block 64	Block 16	Block 32	Block 64
AND	1472	92	46	23	184	92	46	276	138	69
AVG	1664	104	52	26	208	104	52	312	156	78
CND	2104	132	66	33	264	132	66	396	198	99

It is clear that using small block sizes to transfer the files results in an increased number of packets and the values become even more pronounced for multi-hop transactions. When we translate this to the deployment time (Figure 4-16), we can make two interesting observations.

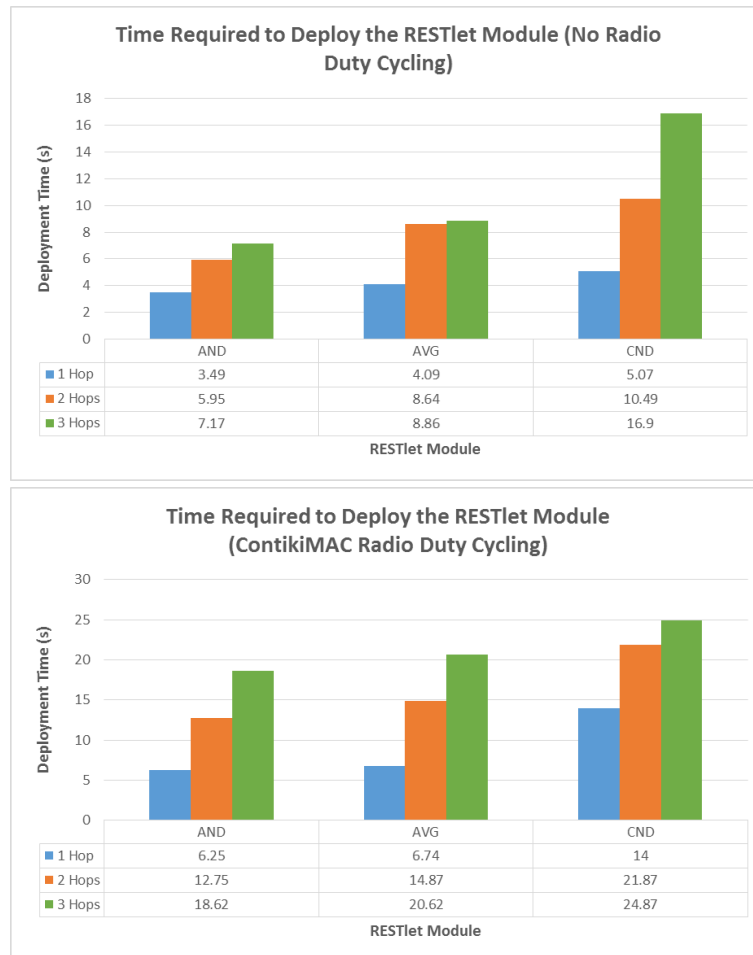


Figure 4-16: Dynamic RESTlet Deployment Time

The first observation, which is rather obvious, is that the time required to transfer the whole dynamic module increases with small block sizes and more hops. Actually, transferring a small block takes less time as compared to transferring larger blocks. But the increased number of packets required to send the complete module, which resulted from using small packets, cancels this advantage and makes using large block sizes preferable. According to the experiment a block size of 64 bytes is ideal for block transfer. If we go beyond 64 bytes, the next possible block size would be 128 bytes. But due to the limits set by the 802.15.4 specification the maximum size of packets at the MAC layer is only 127 bytes

(including headers). So, if the size is more than 127 bytes, the packet will be fragmented leading to suboptimal performance. Due to this reason, the upcoming tests are done using a block size of 64 bytes.

The second observation is the interesting impact of the mechanism used at the underlying radio duty cycling (RDC) layer on the total deployment time. For this test, we used ContikiMAC with channel check rate of 8Hz and NULLRDC at that layer. The ContikiMAC Protocol [4.38] switches off the radio for 99% of the time saving a significant amount of energy that would otherwise be used for passive listening. This means, if the receiver's radio is off when the sender is ready to transmit, it has to wait until the receiver's radio is back on. On the other hand, NULLRDC keeps the radio on all the time allowing immediate transmission and reception of packets. This means, packet transmission through ContikiMAC takes more time than NULLRDC. This clearly explains the shorter deployment time when no RDC is used.

However, this does not necessarily mean that NULLRDC is preferable for dynamic deployment of RESTlets. In fact, ContikiMAC is a better option as demonstrated in the next subsection.

4.5.4.c Energy Consumption

To evaluate the energy usage for transferring dynamic RESTlet modules we used the formula given in [4.39]. According to [4.39], for constrained devices running Contiki, Energy, E , can be computed as,

$$E = E_{CPU} + E_{LPM} + E_{Tx} + E_{Rx} + E_{Idle} + E_{Other}$$

Where

E_{CPU} = Energy consumption of the microcontroller when it is Active

E_{LPM} = Energy consumption of the microcontroller at low power mode (inactive)

E_{Tx} = Energy consumption of the radio transceiver during Transmission

E_{Rx} = Energy consumption of the radio transceiver during Reception

E_{Idle} = Energy consumption of the radio transceiver while sleeping

E_{Other} = Energy consumed by other components such as Sensors and LEDs

The other components are not used for the purpose of this evaluation. Therefore, we exclude it and use the following modified formula to compare energy usage among different options.

$$E' = E_{CPU} + E_{LPM} + E_{Tx} + E_{Rx} + E_{Idle}$$

Energy usage can be defined as power consumption over a period of time. Hence,

$$E' = (P_{CPU} \times T_{CPU}) + (P_{LPM} \times T_{LPM}) + (P_{Tx} \times T_{Tx}) + (P_{Rx} \times T_{Rx}) + (P_{Idle} \times T_{Idle})$$

and power is Voltage multiplied by current drawn by that particular component. For simplicity, we assume that the voltage is uniform and constant for all components while the current drawn is different as per the specification of the component. Hence, we used the following formula to compute energy consumption of the dynamic deployment process.

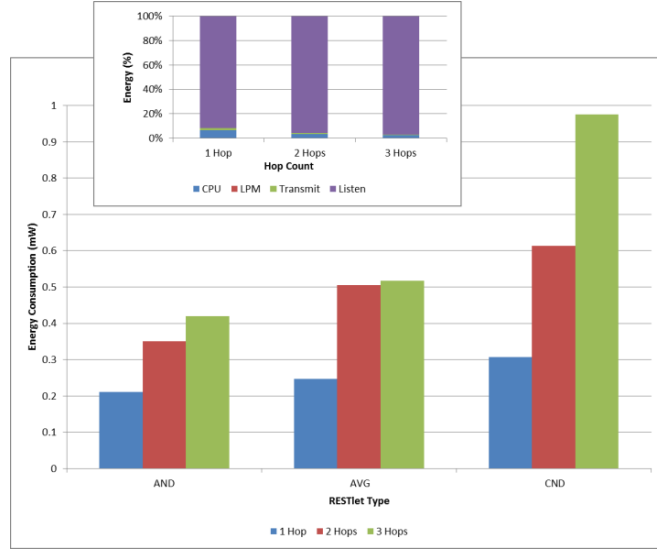
$$E' = (V \times I_{CPU} \times T_{CPU}) + (V \times I_{LPM} \times T_{LPM}) + (V \times I_{Tx} \times T_{Tx}) + (V \times I_{Rx} \times T_{Rx}) + (V \times I_{Idle} \times T_{Idle})$$

The voltage and current values are obtained from Zolertia Z1 datasheet (Table 4-6) and the timing values are obtained through experimentation.

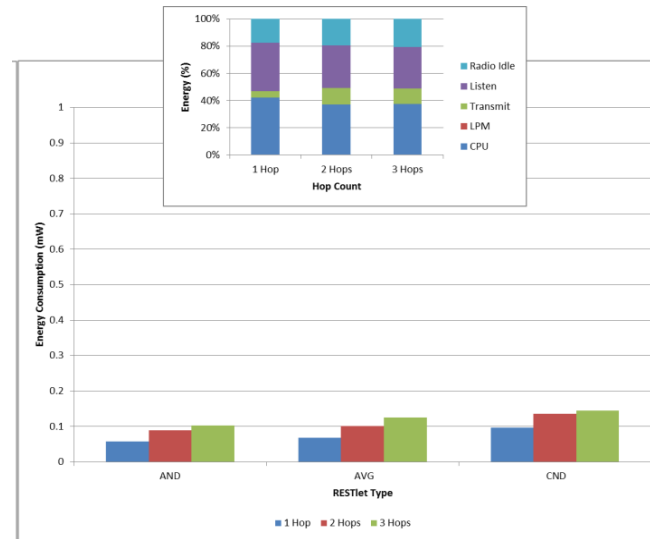
Table 4-6: Power Specification of Zolertia Z1 Mote

Component	Mode	Current (mA)	Voltage
Microcontroller (MSP430)	LPM	0.0000005	3V
	CPU(1MHz)	0.0005	3V
	CPU(16MHz)	0.01	3V
Transceiver (CC2420)	Receive	0.0188	3V
	Transmit	0.0174	3V
	Idle	0.000426	3V
	Power Down	0.00002	3V

Figure 4-17 clearly shows that using no RDC mechanism at RDC layer leads to an increased energy consumption as compared to ContikiMAC in all cases. The small figures inside Figure 4-17 show that the increased consumption is mainly attributed to the passive listening of the radio where the radio is waiting for packets to be transmitted.



(a)



(b)

Figure 4-17: Energy Usage for Dynamic Deployment. a) No RDC Protocol. b) ContikiMAC RDC

4.5.4.d Implementation of Conditional Observation using RESTlets

In [4.18], we made a mathematical computation to demonstrate the reduction in power consumption due to conditional observation as compared to normal

observation. Given an environment where resource state changes every S seconds, the power consumption of normal observation in that period of time, $O(S)$, is given by

$$\begin{aligned} O(S) = & P_{CPU} \times T_{CPU} + P_{LPM} \times (S - T_{CPU}) \\ & + P_{Tx} \times (N \times L) \\ & + P_{Rx} \times (S - N \times L) \times d \\ & + P_{Idle} \times (S - N \times L) \times (1 - d) \end{aligned}$$

If the Max-Age value is set at some arbitrarily higher value for conditional observation, the formula for conditional observation, $CO(S)$, is

$$\begin{aligned} CO(S) = & P_{CPU} \times (T_{CPU} + \Delta_{proc}) + P_{LPM} \times (S - T_{CPU} - \Delta_{proc}) \\ & + p \times [P_{Tx} \times 0 + P_{Rx} \times S \times d + P_{Idle} \times S \times (1 - d)] \\ & + (1 - p) \times [P_{Tx} \times L + P_{Rx} \times (S - L) \times d + P_{Idle} \times (S - L) \times (1 - d)] \end{aligned}$$

In [4.18], we have made protocol modification and were able to alter the behavior of Max-Age values to come up with the formula given above. But, in the current implementation, we do not perform protocol modification and hence the impact of max-age value expiration needs to be considered. Therefore, the formula will be modified to include the energy consumption due to the max-age expiration. In an environment where resource state changes every S seconds, the number of packets, M , transmitted due to Max-age expiration, i.e., irrespective of the resource state change or the transmission criteria, is:

$$M = \text{Floor}\left(\frac{S}{\text{Max_age}}\right)$$

Therefore, $CO(S)$ will be

$$\begin{aligned} CO(S) = & P_{CPU} \times (T_{CPU} + \Delta_{proc}) + P_{LPM} \times (S - T_{CPU} - \Delta_{proc}) \\ & + p \times [P_{Tx} \times (M \times L) + P_{Rx} \times (S - (M \times L)) \times d + P_{Idle} \times (S - (M \times L)) \\ & \quad \times (1 - d)] \\ & + (1 - p) \times [P_{Tx} \times (M + 1) \times L + P_{Rx} \times (S - (M + 1) \times L) \times d + P_{Idle} \\ & \quad \times (S - (M + 1) \times L) \times (1 - d)] \end{aligned}$$

Where,

S = resource state change duration

Δ_{proc} = processing time to evaluate conditions (from prior experiment, this value is 1ms)

p = probability that condition is not fulfilled (hence, no packet transmission)

d = duty cycle (0.01 for ContikiMAC)

L = Length of complete duty cycle (125ms based on ContikiMAC specification)

N = Number of packets transmitted in normal observation in the duration, S

[4.18] gives detailed analysis of energy saving that resulted from using conditional observation. However, by using dynamic deployment of RESTlets for conditional observation, we will consume additional energy for transferring the dynamic module which may result in losing the advantage we got. Table 4-7 shows the energy consumed during transmission. Columns labeled CPU and LPM indicate the energy consumed by the microcontroller in active mode (at 16MHz) and Low Power Mode (LPM), respectively, while the next three columns indicate the energy consumed by the radio for packet transmission and reception and in idle mode as well.

Table 4-7: Energy consumed for Dynamic Deployment

Number of Hops	CPU (Energy)	LPM (Energy)	Transmit (Energy)	Receive (Energy)	Radio Idle (Energy)	Total (mWs)
1Hop	32.309	0.019	4.476	34.260	17.001	88.065
2Hops	40.584	0.030	16.204	42.827	26.584	126.230
3Hops	43.392	0.035	16.579	44.267	30.379	134.651

Here, the good news is that deployment takes place only once, i.e. the moment we decide to host the RESTlet on that particular node. Once deployed, further communications will take place just like the statically created conditional observation modules. This means, after the node starts using conditional observation to send some notification packets, we can start saving energy due to the conditional observe mechanism that results in a reduced number of notifications that needs to be transmitted. This way, we can regain the advantage we lost due to the dynamic deployment. Figure 4-18, shows the time it takes to gain back the advantage of using conditional observation considering the impact of max-age expiration.

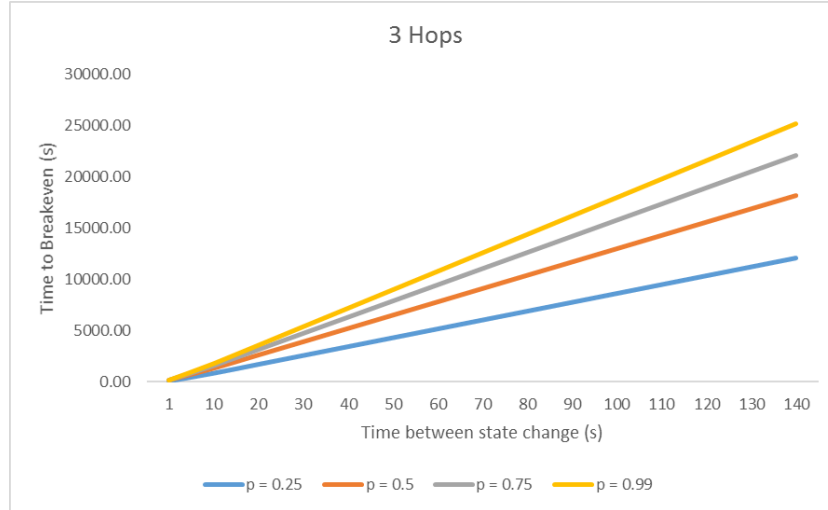


Figure 4-18: Time required to regain power consumed during dynamic deployment

Understandably, the time required to achieve breakeven point increases with the frequency of state change. The effect of p , the probability that the state change does not trigger transmission, also becomes pronounced for cases where state change is not frequent. The increased gap between the data lines towards the right-hand side of the graph suggests this fact. At this point, it is important to note that once the specified number of seconds have elapsed, the node will be saving energy for the rest of its lifetime which is generally measured in months and years.

If the impact of max-age is ignored for conditional observation, we will get a completely different graph (Figure 4-19).

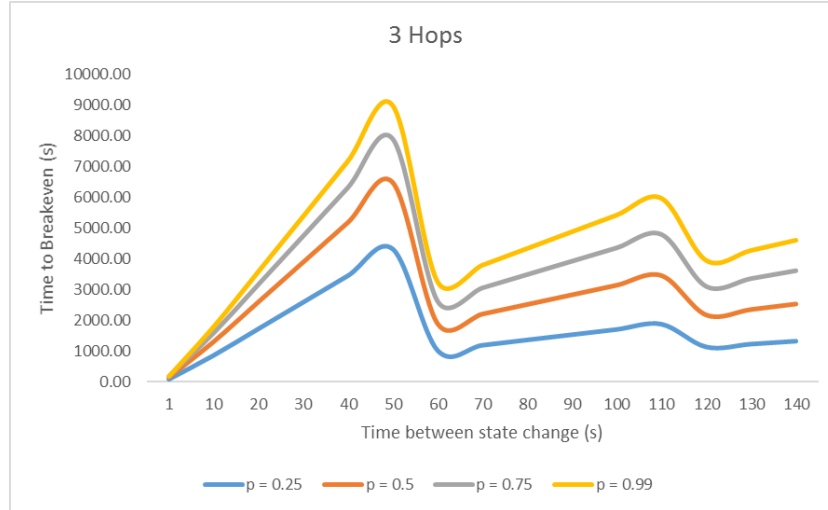


Figure 4-19: Time required to regain power consumed during dynamic deployment (Max-age = 60s)

Just like Figure 4-18, when the probability of sending notifications is low (i.e. low probability that the condition is fulfilled), the time required to reach break-even point is low, especially when the data generation / state change frequency is higher. In the figure, we see that around 60 seconds we see a drastic decrease in required time for break-even. This is attributed to the max-age expiration which triggers generation of another packet in normal observe leading to more energy saving. However, when the resource state change time is higher, the effect of the max-age gets smaller.

4.6 Conclusion and Future work

This paper presented an important extension of our earlier work on RESTlets and Bindings. RESTlets were introduced as CoAP based IoT application building blocks that can be used to develop distributed IoT applications. RESTlets can be linked with sensors to receive inputs and actuators to send their outputs using direct flexible bindings. In our previous work, we showed how statically defined RESTlets can be used for IoT application development. In this paper, we extended this work to introduce dynamic deployment of RESTlets.

We looked at the dynamic deployment of RESTlets on constrained and non-constrained devices separately as the two devices have completely different capabilities. Due to the memory and processing power scarcity of constrained devices, we dynamically deploy only one RESTlet per node which can be instantiated only once but the architecture allows for more RESTlets to be

deployed. On the other hand, non-constrained devices may have dynamic or static RESTlets and may be instantiated multiple times. Dynamic deployment on constrained devices involves the creation of a dynamic module and the transfer of the entire program code into external storage of the constrained node using a bulk transfer method such as CoAP Block-wise transfer. The dynamic loader of the node's firmware will then relocate the code in memory (RAM/ROM) to make it part of the firmware. The deployment process is completed with the creation of CoAP resources for all inputs, outputs and control parameters. This provides greater flexibility to the RESTlet concept by giving an IoT application developer the possibility to select and deploy RESTlets even after the nodes have been deployed in the field. However, this advantage comes with a tradeoff. The first tradeoff is the increased memory requirement due to the inclusion of the Contiki dynamic loader and the file system. But, despite the increase in memory footprint, we have shown that the entire firmware still fits in Class 0 constrained devices. The deployment time and energy consumed for dynamic deployment is another tradeoff. Evaluations performed on Z1 nodes show that, a reasonable sized dynamic module can be deployed in less than 10 seconds on a node after 3 hops (using ContikiMAC). Such a deployment will consume energy, but, in turn, the data processing capabilities offered by the RESTlet may lead to a reduction of the number of packets transferred in the network and thus an overall reduction in energy consumption after some time.

This paper also discussed how our earlier conditional observation mechanism, which was implemented as a CoAP protocol extension, can be implemented by using the RESTlet concept in combination with normal observe. We defined one input, one output and one control parameter per sensor on a node and created a binding relationship between the sensor output and the RESTlet input. The control parameter is used to store the condition types and values. Whenever the sensor output changes, the input of the RESTlet is modified and the processing logic determines if the change needs to be communicated. This way the server side filtering is realized. Even if there is no significant difference in processing time, the dynamic loading introduces some wastage of resources which may counter-balance the energy saved by conditional observation (as compared to normal observation). However, we showed that after a few notifications, the conditional observe method starts saving energy.

Still the proposed solutions can be further optimized. One of the ideas we will be looking at is cross-layer optimization options. Optimal placement of RESTlets is another potential area for improvement. Providing an easy to use programming interface is also a possible future work. By providing visual programming tools, we may reduce RESTlet-based IoT application development to a simple drag and drop operation.

Acknowledgement

The scholarship of the corresponding author is covered by VLIR-UOS through the collaboration with Jimma University in the IUCJU program.

References

- [4.1] A. Wood, "The internet of things is revolutionizing our lives, but standards are a must," *The Guardian*, 2015. [Online]. Available: <https://www.theguardian.com/media-network/2015/mar/31/the-internet-of-things-is-revolutionising-our-lives-but-standards-are-a-must>. [Accessed: 14-Sep-2016].
- [4.2] ITU-T, "Recommendation ITU-T Y.2060: Overview of the Internet of things." International Telecommunication Union, Geneva, 2013.
- [4.3] K. Rose, S. Eldridge, and C. Lyman, "The internet of things: an overview," *Internet Soc.*, no. October, p. 53, 2015.
- [4.4] F. Orlando, "Gartner Identifies the Top 10 Strategic Technologies for 2012," *Gartner Inc.*, 2011. [Online]. Available: Gartner Identifies the Top 10 Strategic Technologies for 2012. [Accessed: 15-Sep-2016].
- [4.5] F. Orlando, "Gartner Identifies the Top 10 Strategic Technology Trends for 2013," *Gartner Inc.*, 2012. [Online]. Available: <http://www.gartner.com/newsroom/id/2209615>. [Accessed: 16-Sep-2016].
- [4.6] F. Orlando, "Gartner Identifies the Top 10 Strategic Technology Trends for 2014," *Gartner Inc.*, 2013. .
- [4.7] F. Orlando, "Gartner Identifies the Top 10 Strategic Technology Trends for 2015," *Gartner Inc.*, 2014. [Online]. Available: <http://www.gartner.com/newsroom/id/2867917>. [Accessed: 01-Sep-2016].
- [4.8] F. Orlando, "Gartner Identifies the Top 10 Strategic Technology Trends for 2016," *Gartner Inc.*, 2015. [Online]. Available: <http://www.gartner.com/newsroom/id/3143521>. [Accessed: 16-Sep-2016].
- [4.9] C. Stamford, "Gartner Says 6.4 Billion Connected 'Things' Will Be in Use in 2016, Up 30 Percent From 2015," *Gartner Inc.*, 2015. [Online]. Available: <http://www.gartner.com/newsroom/id/3165317>. [Accessed: 16-Sep-2016].
- [4.10] Dave Evans, "The Internet of Things," *Cisco Inc.*, 2011. [Online]. Available: <http://blogs.cisco.com/diversity/the-internet-of-things-infographic>. [Accessed: 16-Sep-2016].
- [4.11] Ericsson, "More Than 50 Billion Connected Devices," *White Pap.*, no. February, pp. 1–12, 2011.

- [4.12] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "RFC 4944: Transmission of IPv6 Packets over IEEE 802.15.4 Networks." IETF, pp. 1–30, 2007.
- [4.13] T. Winter, P. Thubert, A. R. Corporation, and R. Kelsey, "RFC6550: RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks." IETF, pp. 1–157, 2012.
- [4.14] Z. Shelby, K. Hartke, and C. Bormann, "RFC 7252: The Constrained Application Protocol (CoAP)." IETF, pp. 1–112, 2014.
- [4.15] Z. Shelby, "Embedded web services," *IEEE Wirel. Commun.*, vol. 17, no. 6, pp. 52–57, 2010.
- [4.16] L. Richardson and S. Ruby, *RESTful Web Services*, First. O'REILLY, 2007.
- [4.17] S. Li, K. Li., J. Hoebeke, F. Van den Abeele, and A. Jara, "Conditional observe in CoAP." IETF, 2014.
- [4.18] G. Teklemariam, J. Hoebeke, I. Moerman, and P. Demeester, "Facilitating the creation of IoT applications through conditional observations in CoAP," *EURASIP J. Wirel. Commun. Netw.*, vol. 1, no. 1, 2013.
- [4.19] M. Kovatsch, "Firm firmware and apps for the internet of things," *Proceeding 2nd Work. Softw. Eng. Sens. Netw. Appl. - SESENA '11*, p. 61, 2011.
- [4.20] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things," *2010 IEEE/IFIP Int. Conf. Embed. Ubiquitous Comput.*, pp. 347–352, 2010.
- [4.21] D. Alessandrelli, M. Petracca, and P. Pagano, "T-Res: Enabling reconfigurable in-network processing in IoT-based WSNs," *Proc. - IEEE Int. Conf. Distrib. Comput. Sens. Syst. DCoSS 2013*, vol. 317671, pp. 337–344, 2013.
- [4.22] J. Liu, J. Reich, and F. Zhao, "Collaborative in-network processing for target tracking," *EURASIP J. Appl. Signal Processing*, vol. 2003, no. 4, pp. 378–391, 2003.
- [4.23] E. FASOLO, M. ROSSI, J. WIDMER, and M. Zorzi, "IN-NETWORK AGGREGATION TECHNIQUES FOR WIRELESS SENSOR NETWORKS: A SURVEY," *IEEE Wirel. Commun.*, no. April, pp. 70–87, 2007.
- [4.24] A. Azzara and L. Mottola, "Virtual resources for the Internet of Things," *IEEE World Forum Internet Things, WF-IoT 2015 - Proc.*, pp. 245–250, 2016.

- [4.25] C. Perera, P. P. Jayaraman, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Sensor Discovery and Configuration Framework for The Internet of Things Paradigm," 2013.
- [4.26] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "DIANE – Dynamic IoT Application Deployment," 2015.
- [4.27] G. Teklemariam, F. Van den Abeele, I. Moerman, P. Demeester, and J. Hoebeke, "Bindings and RESTlets: A Novel Set of CoAP-Based Application Enablers to Build IoT Applications," *Sensors (Basel)*, vol. 16, no. 8, 2016.
- [4.28] K. Hartke, "RFC 7641: Observing Resources in the Constrained Application Protocol (CoAP)." IETF, pp. 1–30, 2015.
- [4.29] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Futur. Gener. Comput. Syst.*, vol. 29, no. 7, pp. 1645–1660, 2013.
- [4.30] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A RESTful runtime container for scriptable internet of things applications," *Proc. 2012 Int. Conf. Internet Things, IOT 2012*, pp. 135–142, 2012.
- [4.31] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - A lightweight and flexible operating system for tiny networked sensors," *Proc. - Conf. Local Comput. Networks, LCN*, pp. 455–462, 2004.
- [4.32] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A low-power CoAP for Contiki," *Proc. - 8th IEEE Int. Conf. Mob. Ad-hoc Sens. Syst. MASS 2011*, pp. 855–860, 2011.
- [4.33] C. Bormann and Z. (Ed. . Shelby, "RFC 7959: Block-Wise Transfers in the Constrained Application Protocol (CoAP)." IEEE, pp. 1–37, 2016.
- [4.34] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," *Proc. 4th Int. Conf. Embed. networked Sens. Syst. - SenSys '06*, p. 15, 2006.
- [4.35] Zolertia, "Z1 Datasheet," pp. 1–20, 2010.
- [4.36] Texas Instruments, "MSP430x2xx Family USER ' s Guide." 2013.
- [4.37] C. Bormann, M. Ersue, and A. Keranen, "RFC 7228: Terminology for Constrained-Node Networks." IETF, pp. 1–17, 2014.
- [4.38] A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol," *SICS Tech. Rep. T201113*, ISSN 1100-3154, pp. 1–11, 2011.
- [4.39] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-based on-line energy estimation for sensor nodes," *Proc. 4th Work. Embed. networked sensors - EmNets '07*, p. 28, 2007.

5

Transparent Recovery of Dynamic States on Constrained Nodes through Deep Packet Inspection

One of the fundamental components of IoT applications are smart objects which are usually characterized by several resource constraints. Due to these constraints and other maintenance activities, these devices may have to go through a power cycle. When they come back online, they will lose all data that was created dynamically during runtime as a result of interactions with other nodes and that was stored in volatile memory. These dynamic states must be recovered for the IoT application to operate as required. In this chapter, we discuss a transparent dynamic state recovery mechanism that makes use of deep packet inspection. A state directory is built at the LLN gateway by dynamically intercepting every packet that traverses the gateway. The state directory entry contains all the necessary information that can be used to regenerate any dynamic state on a node. When a node boots up, the state directory is consulted and the packets that previously created the dynamic states will be replayed. This process is done transparently without the involvement of the communicating parties.

Girum Ketema Teklemariam, Floris Van den Abeele, Ingrid Moerman, Piet Demeester, Jeroen Hoebeke. *Transparent Recovery of Dynamic States on Constrained Nodes through Deep Packet Inspection*. Submitted to Journal of Sensors, December 2017

Abstract: *Many IoT applications make extensive use of constrained devices which are characterized by unexpected failures. In addition, nodes could be put offline temporarily for maintenance (e.g. battery replacement). These incidents lead to loss of dynamic data that is generated due to the interactions between nodes. For instance, configuration settings adjusted by sending PUT request to the sensor will be lost when the node is rebooted. The lost data, which we call dynamic states, leads to erroneous results or malfunctions of the IoT application. In this paper, we introduce an intelligent dynamic state recovery mechanism through deep packet inspection. A State Directory, placed at the gateway, intercepts every communication between an external device and constrained devices and stores (or updates) information that is important to restore dynamic states. When a node reports a reboot, the state directory replays the packets that generated the dynamic state so that all dynamic states are restored. We implemented the solution on a non-constrained device that acts as a gateway to the constrained network and tested the results using Cooja simulator.*

5.1 Introduction

IoT applications heavily depend on the interconnection of smart objects through wireless links. The smart objects are characterized by limited capabilities such as memory, power supply, processing power and communication bandwidth. Due to these constraints, the communication network is characterized by unstable links. In combination with the low power aspect, these networks are often referred to as Low-Power and Lossy Network (LLN). Usually, LLNs are connected to external networks, ultimately to the Internet, through gateways. This architecture allows IoT application components to reside entirely in the LLN or to be distributed across different locations (in the LLN, at the gateway and in the cloud). In such models, components of the IoT application residing outside the LLN need to interact with components residing inside. For instance, in a home automation application, a user may adjust the house temperature using his/her smartphone from the Internet. The client is residing outside the LLN while the temperature sensor and actuator are inside.

Different approaches can be used to enable interactions among the different components of the IoT application. One such approach is the use of CoAP-based communication between clients and servers. The Constrained Application Protocol (CoAP) is a light-weight HTTP-like protocol that uses the same GET, PUT, POST and DELETE methods to access and update resource representations hosted on servers [5.1]. Unlike HTTP, CoAP uses UDP at the transport layer to avoid the overhead introduced by TCP. However, the protocol offers reliability through Confirmable messages. An important extension of CoAP, named Observe [5.2], is also used in IoT applications involving monitoring of resource states. This

protocol lets clients register their desire for updated information from the server. After successful registration, the server sends every resource state change to the client until the relationship expires or is proactively terminated by the client. As such, the data collection part of IoT applications can be developed by programming the different components entirely using CoAP and its observe extension.

Such interactions may generate *dynamic states* that have to be stored in the volatile memory of the constrained device. In addition, other dynamic states such as the list of clients which established observation relationships with the server or bindings [5.9] are also stored in the volatile memory. All dynamically created states will be lost in case the node reboots for any reason. A reboot can be triggered by a crash or can be the consequence of a firmware update. Such reboots may occur and necessitate the recovery of the application state in order to ensure continuity of the applications outside of the network. This can be achieved by letting the application detect the failure and reinstall the state. This process can be time-consuming and must be managed properly by the applications. Therefore, a fast and automatic recovery mechanism that can deal with crashes in a way that is transparent for the applications is very relevant.

In this paper, a novel approach for crash recovery is introduced. We introduce the concept of a *State Directory* at the LLN gateway which automatically intercepts and inspects all interactions between clients and servers residing at either side of the gateway. Besides monitoring all application-layer interactions with the LLN, it is able to store all dynamic states generated on the constrained nodes. Upon reception of a start-up message from a node, the state directory replays the requests that generated the dynamic states to recover them on the server. Four types of interactions from external clients that may create dynamic states are intercepted. These are, PUT requests that modify configuration parameters, observation requests, binding requests and dynamic loading requests.

The remainder of the paper is organized as follows. The next section discusses related work in the area of the recovery of nodes and resources. Dynamic states and state recovery is discussed in more detail in Section three. Dynamic state recovery options, including our novel transparent dynamic state recovery mechanism, are explained in section four, while Section five describes the implementation of the proposed solution. Functional and performance evaluation of the proposed mechanism is given in Section six. Section seven concludes the paper.

5.2 Related Work

Many IoT applications that involve embedded sensors and actuators rely on LLNs formed by the interconnection of constrained devices. A failure of a constrained device that results in a reboot may affect the entire operation of the application. Several attempts have been made to reduce the impact of such failing nodes on the accuracy of the applications [5.3]–[5.5]. Most of these works focus on finding alternative nodes or links in order to avoid malfunctioning. In contrast, our work focuses on regenerating dynamic states on nodes in order to resume their function when they are back online. This way, the IoT application that depends on the installed state is able to continue its operation without taking any specific actions. The aforementioned works focus completely on failed/failing nodes or routes.

Another approach, which has similarities to ours, focuses on storing dynamic states at a location from where it can be restored. The first notable work in this category is given in [5.6]. The paper presents a sensor network model in which each node stores sensor data locally and provides a database query interface to the data. Each sensor device runs its own database system using Antelope. Antelope provides a dynamic database system that enables runtime creation and deletion of databases and indices. Antelope uses energy-efficient indexing techniques that significantly improve the performance of queries. This technique may be used for storing state information in neighboring constrained devices and can be used to restore the states when the node comes back from failure. However, this approach is different from ours in many aspects. Firstly, the approach needs a database system and an interface to interact with the database on the constrained node, which is expensive for the node. Secondly, the info is stored on a constrained node which is also prone to failure. We select the gateway to store the state information to avoid this issue. Finally, the state information collection is done exclusively using database query interfaces which uses either push or pull methods, whereas our approach is fully transparent to both the client and the server. Yet another notable work is LUSTER [5.7]. LUSTER presents a hierarchical architecture that includes distributed reliable storage, delay-tolerant networking and deployment time validation techniques. Fault-tolerant storage is provided by discretely listening to sensor node communications without the need of dedicated queries. This is realized through an overlaid, non-intrusive reliable storage layer that provides distributed non-volatile storage of sensor data for online query, or for later manual collection. The storage layer is used for storing sensor data. The permanent storage that is used to store dynamic data is the only similarity of this approach to ours. However, LUSTER uses a storage layer while we use the gateway for permanent storage.

5.3 Dynamic States and State Recovery

5.3.1 Dynamic States

IoT applications that make use of embedded web services often involve one or more constrained devices that generate or consume data inside a Low-Power and Lossy Network (LLN). Sensor nodes usually generate data to be consumed by another constrained node or by a non-constrained device residing outside the constrained network. Similarly, actuator nodes consume data that is generated inside the LLN by a sensor node or by a non-constrained device on the Internet. As such, interactions between different components need to take place to realize the desired functionality of IoT applications. In most, if not all, cases, these interactions originate from external devices such as smartphones or monitoring stations. In rare cases, interactions may also originate from the LLN nodes if they are pre-configured to interact with each other. Such interactions may result in the generation of new state information that is either used immediately or stored at the constrained device for future use. For instance, a simple GET request sent to a sensor node from a smartphone usually results in the generation of sensed data and the immediate transmission of responses containing that data. Such information does not need be stored in the LLN as subsequent requests will result in the generation of new data values that will be communicated back. On the contrary, PUT requests that modify the operation threshold of a temperature sensor result in information that needs to be stored locally. We call such information *Dynamic States*. At this juncture, it is important to distinguish between dynamic states and sensor resource states. Sensor resource states are sensor readings exposed as a CoAP resource to be accessed by clients while dynamic states are any set of information generated as a result of interaction between sensors, actuators and other devices and are stored in memory.

Various approaches can be followed to recover lost state information, but all solutions involve storing duplicate information, preferably, on a more permanent or reliable storage. We call these entities *State Directories (SD)*. A State Directory (SD) can be defined to collect and store all dynamic states at a central location. The state directory is then referred and updated regularly by intercepting every potential interaction between devices that may generate or modify dynamic states. The interception may be done by a device that is powerful enough to store and process dynamic states and that has access to the traffic flow without much overhead. Considering this, the LLN gateway may be a good candidate to host the SD and intercept the traffic.

Interactions that result in data that needs to be stored are of great importance for IoT applications. Some of these interactions are:

- A. Parameter Modification at Runtime** – PUT requests sent to nodes in the LLN usually affect some parameters of the node. Such information needs to be stored in memory so that the node can operate as per the new requirement. Examples of such information include alteration of operation thresholds, control parameter modifications and actuations. Every request may change a default value generating a new dynamic state or update the existing one.

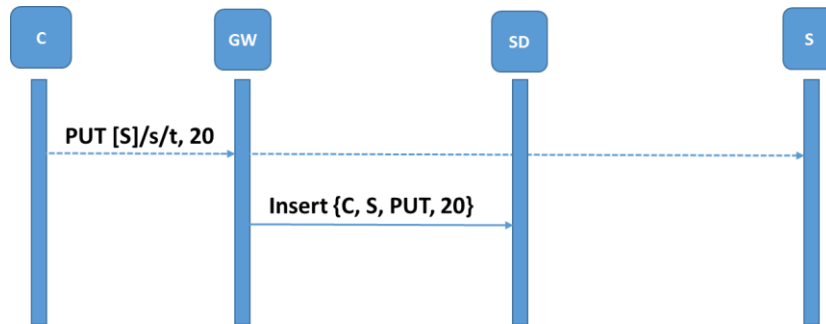


Figure 5-1: CoAP Interaction - PUT Request

Figure 5-1 shows a PUT request sent to the $/s/t$ resource of the sensor **S** to change the value to 20. On its way to the destination, the gateway (GW) intercepts the packet and stores the client and server information along with the type of entry and the value in the SD.

- B. Observation Request** – In monitoring applications, clients send observation request to sensors to be registered as observers so that an up-to-date representation is sent to them as soon as it becomes available. After receiving the request, the sensor node stores the details of the client for future notifications. If conditional observe is used, as described in [5.8], notification criteria will also be stored at the sensor. In addition to this, subsequent notifications may also update the dynamically generated states (Figure 5-2).

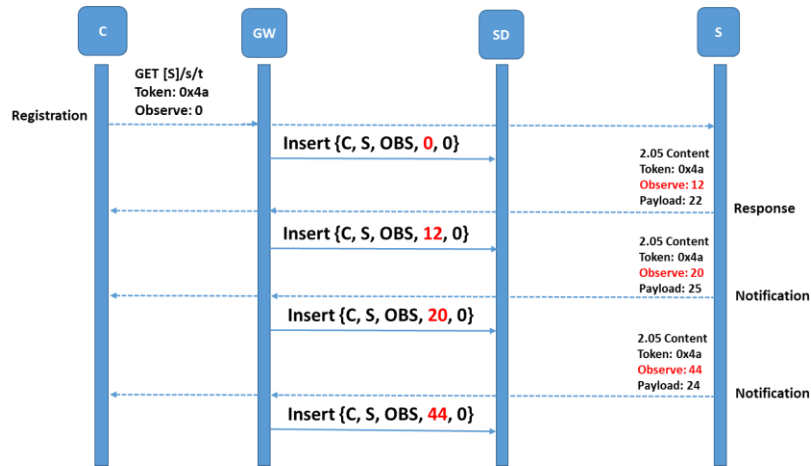


Figure 5-2: CoAP Interaction - Observation Request

Both Observation registration requests and notifications generate dynamic states. In Figure 5-2, the registration packet results in the generation of a new entry in the SD. In this case, the observe counter, set to 0, and the retransmission counter (also set to 0) along with the client and server information are stored. Subsequent notifications are also shown, updating the observe counter from 0 to 12, 20 and 44, respectively. The retransmission counter is updated only if there are retransmissions that will lead to the cancellation of the relationship when the maximum number of retransmissions is reached.

- C. Flexible Binding Relationship Creation** – A flexible binding relationship is an observation relationship between devices that is established by a third party device [5.9]. Bindings are established by devices residing outside the LLN and notifications may be sent to nodes in the LLN. Unlike observation relationships, bindings generate new PUT requests instead of responses for the original GET request. The information on how to generate the PUT request needs to be stored by the node.

Only the binding request that is sent from the external device is intercepted by the GW (Figure 5-3). Binding requests are identified by the BIND_INFO included in the GET request. As the binding information is required to recover the state later on, this information is stored in the state directory. Notifications generated by the sensor and sent to the actuator are not intercepted and have no impact on the information stored in the SD.

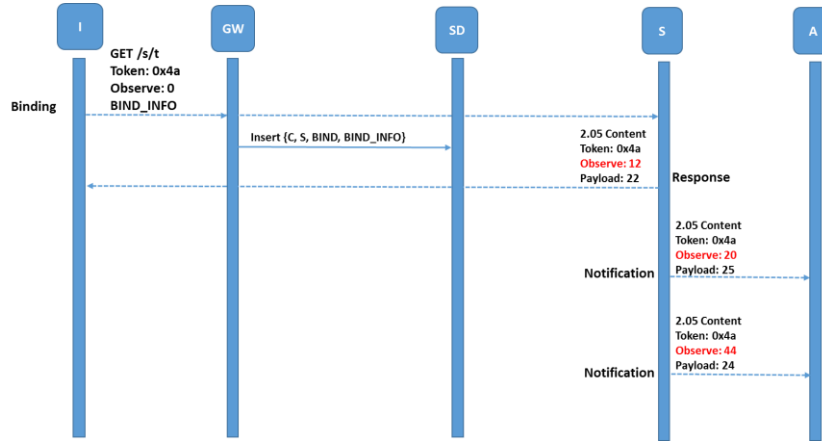


Figure 5-3: CoAP Interaction - Binding Request

D. Runtime Deployment of Application Code – Some node architecture allow the deployment of code at runtime [5.10]. The dynamic code may range from small bug fixes to replacements of a portion of the firmware [5.11]. Runtime deployment can also be used to provide processing capacity to constrained nodes. Dynamically deployed RESTlets [5.10] are examples of code fragments that are deployed at runtime. In many cases, the dynamic component is first sent to a permanent storage (e.g. external flash) on the node before it is relocated and loaded into the main memory. Even if the file containing the raw code is in external storage, the relocated, ready-to-use component is stored in memory and can be referred to as dynamic state information. Once loaded into the memory, there are no further interactions that modify the stored information unless it is a new update, which is treated as a new request that replaces the old information.

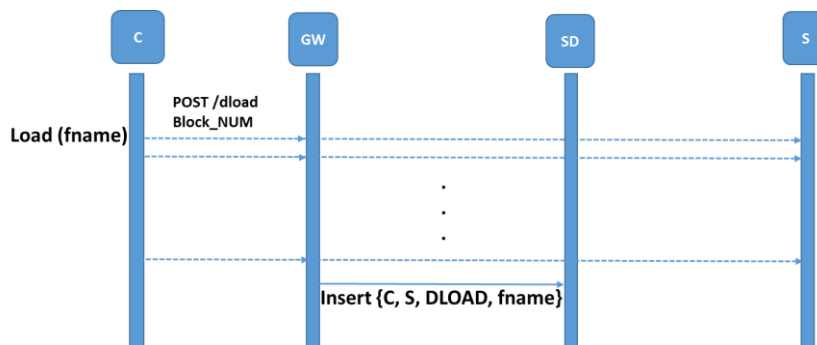


Figure 5-4: CoAP Interaction – Runtime Deployment

As shown in Figure 5-4, dynamic modules are transferred using CoAP Block-wise transfer. Once all the blocks have been received by the node, the

filename will be stored on the GW together with other relevant information. This will enable the device to dynamically load the module from external memory of the node. Alternatively, we may store the entire dynamic code locally in the SD and replay the transfer of the entire module from the SD.

5.3.2 Recovery of Dynamic States

The states generated through the aforementioned interactions are stored in the memory of the constrained nodes. In this work, we consider state information that is stored in volatile memory and that is lost after rebooting the node. So, in case the constrained nodes go through a power cycle due to an internal error (e.g. software error) or are temporarily put offline for maintenance (e.g. battery replacement), the nodes have lost all dynamic states when they come back online. This situation affects all IoT applications that rely on the previously installed state in that node. A structured and efficient state restoration mechanism is crucial to ensure that all dynamic state information is captured and updated and the latest state is restored when the node comes back to life. Figure 5-5 shows an example of a structured state restoration process.

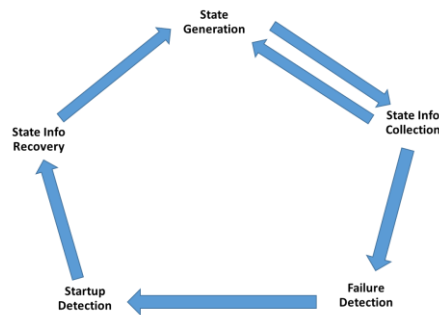


Figure 5-5: Dynamic State Restoration Cycle

1. **State Generation** – As explained above, several nodes involved in IoT applications generate dynamic states as a result of interactions with other constrained or non-constrained nodes. Some interactions generate new state information every time, while others continuously update the dynamically created state. For instance, every observation relationship request creates a new dynamic state, whereas notifications generated due to resource state changes update part of the already existing dynamic state (e.g. the latest observe counter).
2. **State Information Collection** – In order to successfully recover the dynamic states, it is imperative to carefully collect the states as soon as they are generated and/or modified. Some interactions create new states and others modify existing ones. Some interactions may also result in the removal of the dynamic state (For instance, a GET request with the

observe option set to 1). Therefore, it is vital to distinguish between these interactions and take the appropriate action. Collection of state information must be done transparently without human intervention and knowledge of the involved parties.

3. **Failure Detection (Optional)** – A node might not be available for some time due to physical failure or link failure. In some cases, failure detection is important to defer re-registration (or cancelation) of relationships by clients due to the unavailability of sensors for a short period of time. The best example of such a scenario is an observation relationship established between a client and a sensor node with a fixed Max-Age value. If the node takes more time than the Max-Age value to get back online, the client sends a new GET request to re-register its interest or even a RST to actively cancel the relationship. The re-registration (and possible cancelation) may be avoided in case the failure is detected earlier and an intermediary can respond to the requests in place of the sensor. Early detection may also allow intermediary devices, such as the LLN gateway, to play a role in establishing a separate relationship with replacement devices if the original sensor is gone forever [5.12]. Once the replacement is put in place having the same IP address, the intermediary may store all states of the old node into the new node so that the transaction continues as before. If such intervention is not required, failure detection is not mandatory, startup detection is.
4. **Startup Detection** – A node that is not available for some time does not necessarily imply that it has physically failed. Possible link failures on the path between the sensor and the client can break the communication. Timely startup detection is a vital step in state information recovery. A mechanism must be in place to differentiate between physical node failure and link failure to properly recover state information, as link failures typically do not require recovery.
5. **State Information Restoration** – This is the last step that puts all the dynamic state information back into the memory of the node. Since all transactions are expected to continue as before, we must make sure that all states are restored before the client is aware of the brief absence of the constrained node. In addition, we must make sure that the recovery procedures do not lead to the generation of too much communication creating congestion at specific nodes in the network leading to further timeouts in other communications.

5.4 Dynamic State Recovery

As described in the previous section, IoT applications that make use of constrained objects usually depend on dynamically generated states. The loss of such states

negatively impacts the performance and/or accuracy of the application. This very idea makes state recovery an important component of IoT applications.

As mentioned earlier, one of the important steps in dynamic state recovery is the collection of the state information in a way that is easy to recover. Dynamic state collection can be done by the node itself using a push mechanism, where the node sends every new or updated state to a pre-configured or negotiated state directory as soon as it is created. The major drawback of this mechanism is the additional overhead it creates in the constrained network, in order to store the states at the SD. Every transaction that gives rise to new or updated dynamic state information results in the generation of a packet towards the SD. This may lead to various problems including an increased power consumption as well as an increased number of packets inside the LLN that results in network congestion. The opposite of this approach is a pull mechanism initiated by the SD. This can be done either by continuous polling or using publish/subscribe mechanisms such as CoAP observe. In both cases, the approach suffers from the same drawback as the push mechanism.

An innovative and less expensive approach is to do the state collection in a way that is transparent to both the client and the server. This can be done by intercepting and inspecting the packets as they traverse the network towards their destination and deciding whether they will result in a new or updated dynamic state. As the interactions that affect the stored dynamic states of a node are known, it is easy to anticipate the change and store the appropriate information in the SD.

Such methods generally work well for unencrypted transactions. Encrypted messages cannot be easily inspected and require another approach. However, this can be overcome by using a trusted gateway and sensor virtualization as shown in [5.12]. The next two subsections describe in more detail the transparent state recovery for both unencrypted and encrypted communications, followed by a subsection on alternative approaches.

5.4.1 Transparent Dynamic State Recovery for Unencrypted Communication

An essential aspect for this approach to work is the location of the state directory. Most interactions, if not all, that alter dynamic states of a constrained device originate from the non-constrained network such as the Internet. Such packets must pass through the LLN gateway before they reach the constrained node. Similarly, responses also go through the gateway on their way to the external device. Moreover, the LLN gateway is a non-constrained device that is always on and can handle the real-time interception and inspection of all packets to and from

the LLN. All these features make the LLN gateway an ideal location to place the SD. Our transparent dynamic state recovery solution is based on this basic idea. The gateway intercepts all the traffic between the LLN and the external network and stores all relevant information needed to recreate the dynamic states at a later time when recovery is required (Figure 5-6). As shown in the figure, a PUT request sent from a smartphone to a constrained node residing in the LLN passes through the gateway, which also houses the state directory. The state directory contains a list of values required to regenerate all dynamic states on the constrained devices.

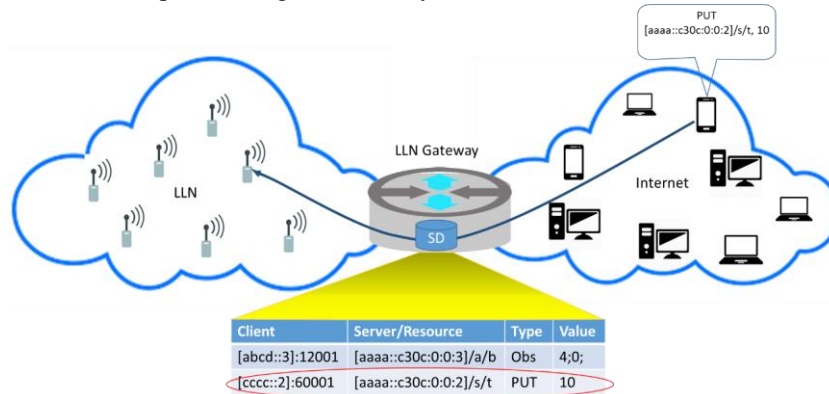


Figure 5-6: Placement of State Directory at the LLN Gateway

Other mechanisms could be incorporated to our solution in order to capture interactions originating and terminating in the LLN, but this is outside the scope of this work. The different steps involved in the transparent dynamic state recovery, as introduced in Figure 5-5, and how they can be realized are discussed below.

5.4.1.a SD – Node Association

The first step in the transparent dynamic state recovery mechanism is establishing a relationship between the state directory and the sensor nodes inside the LLN. This step is important for subsequent steps too. This can be done in several ways.

Proactive Registration: one possible method is to allow explicit registration of nodes upon startup. Whenever a node boots up, it is expected to send a packet to the gateway, which registers the node as a potential node that may store dynamic states. Hereafter, any request-response interaction that may modify the dynamic states on this node will be intercepted. In addition, the registration request can also be used to detect the startup of the node and may trigger the state recovery process. In this case, the packets intercepted will be the ones sent to and from the registered nodes. The main drawback of this approach is that if the registration request is lost

in between, the gateway will not be able to know the existence of this node. However, this can be overcome by sending confirmable registration requests.

Reactive Registration: by default, the gateway inspects all packets entering the LLN. The moment the first packet is sent towards a node inside the LLN, the node association is created. Further, the packet is inspected and, if needed, the gateway stores the dynamic state provisionally and starts a timer. If an error message is sent back from the LLN stating absence of the node or unwillingness to comply to the request before the timer expires, the provisional information will be removed from the SD. However, if a positive response is intercepted or the timer expires, the provisional status will be changed to permanent. All further packets to and from that node will now be intercepted. This is a completely transparent mechanism where neither the client nor the sensor are aware of the existence of the SD. In addition, no prior registration is required for the operation. However, if an error message is generated and gets lost on its way to the gateway, the gateway will still store a state that does not exist, resulting in inconsistent information. This is the major downside of this method. In addition, we need another mechanism to detect rebooting of the nodes as the approach does not provide an inherent way to detect startups.

Inference from the LLN Routing Table: in RPL based networks, the root of the DODAG stores all available nodes in the LLN. The LLN gateway, being closest to the RPL root, may just store all available nodes by collecting the route information from the root node. This can easily be achieved if the root node exposes the route information as a CoAP resource to the gateway and the gateway registers as an observer. The gateway stores all the nodes in the LLN irrespective of their capacity and role and packets to/from these nodes are intercepted. Also nodes that are not involved in any dynamic state generating interaction (such as simple routers) are stored at the SD. This approach suffers from multiple drawbacks. First, re-registration of a node at the DODAG root does not necessarily mean the node is rebooting. When a link to the parent fails, a node chooses a new route and is re-registered at the DODAG root. This information reaches the SD giving it an incorrect information that the node is coming back up from failure. Second, when a node or a link fails, all nodes that use the failed link or node will find an alternate route and notify the DODAG root. Again, this gives inaccurate information to the SD informing it to start the recovery process. Finally, every node sends keep-alives to the root periodically which are treated as updates.

5.4.1.b State Information Collection

State information collection is done to ensure that the correct dynamic state is captured in the state directory. As mentioned before, all requests with the potential of updating a dynamic state must be intercepted so that the SD information is updated. Figure 5-7 shows how the dynamic state collection works.

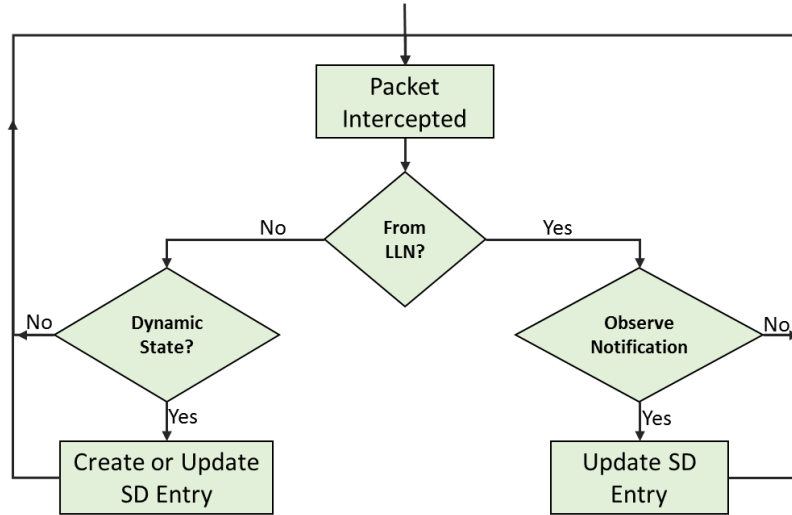


Figure 5-7: Dynamic State Information Collection

Every packet that passes through the gateway is intercepted and different actions are taken based on the origin of the packet. If the packet originates from the LLN, the SD entry associated with the content of the intercepted packet will be updated only if the packet is an observe notification. However, if a request with the potential of creating or modifying a dynamic state on a node in the LLN is intercepted from the Internet, an existing entry will be updated or a new one is created on the SD. As mentioned before, the intercepted requests from the Internet are PUT requests, observe requests, binding Requests and dynamic deployment requests.

A PUT request from the external network triggers the gateway to check the SD for a matching request to the destination. If it exists, the value will be updated. Otherwise, a new entry will be created for the destination. For instance, if a user updates the operating temperature of a thermostat multiple times, the first request creates the information in the SD and subsequent requests will update it.

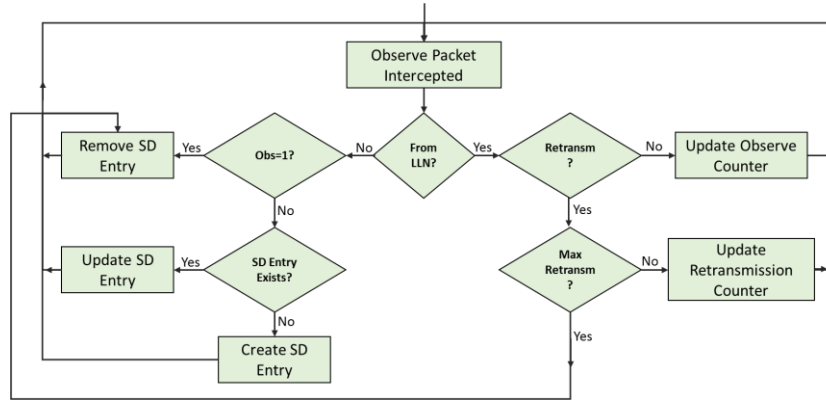


Figure 5-8: Observe Request Information Collection

Observation requests require more thorough examination of the packet before the gateway decides how to update the SD (Figure 5-8). Observe request from the external network with an observe value different from 1 will establish a new observe relationship between the sender and the LLN node. Accordingly, the gateway creates a new entry in the SD. In case there is already an existing observe relationship between these two parties, this will be indicated by the existence of an entry in the SD. Then, the information will be updated as the same will happen at the LLN node when it receives another observe request from the same client to the same resource. On the other hand, an observe request from the external network with the observe value set to 1, removes an existing relationship and hence is implied by removing an existing entry from the SD. All packets that originate from the LLN that include the observe option are notifications generated as a result of resource state changes affecting an existing observe relationship. So, the gateway captures the packets and updates the current entry in the SD. This is particularly important for two reasons. One, if the node reboots, after successful dynamic state recovery, the observe value used in subsequent notifications must be a continuation of the last notification. Otherwise, the client will ignore the value as obsolete eventually leading to Max-age expiration and, consequently, to re-registration or cancellation of the relationship. The second reason is to capture timeouts. Every now and then, the sensor obliges the client to acknowledge reception of notifications. However, if the client is not available to do so, the sensor retransmits the packets for a fixed number of times and removes the observe relationship in case it fails to get any acknowledgement. This means, the entry should also be removed from the SD. By examining the packet, we detect retransmissions and hence can remove the SD entry after the maximum number of retransmissions is reached.

Binding relationships are established by sending packets from the external network to the LLN. The packets contain the observe option along with the four binding options explained in [5.10]. Any binding request must be intercepted and the appropriate information stored in the SD. Unlike the regular observe operation, subsequent transactions, except the response to the first request, do not reach the gateway. Therefore, no update will be made for binding relationships after the relationship is established. In order to capture events happening inside the LLN, e.g. cancellation of the binding relationship by either party, different mechanisms may be applied. For instance, the gateway may regularly check the binding directory of the sensor node to see if all bindings are intact.

Requests sent to dynamically loaded RESTlets, can be identified by the block transfer option sent to the dynamic loader resource of the constrained node. As far as storing information in the SD is concerned, we have two options here. The first option is capturing all blocks and storing the dynamic module at the gateway so that recovery can be done by resending the dynamic module to the constrained node. The other option will be storing only the filename and the node's address at the SD. But for this to work, the dynamic module must be available at a permanent storage of the node. Upon reboot, the gateway will send a packet to the node to read and load the dynamic module from the storage.

5.4.1.c Failure Detection

Failure detection is optional and enables the gateway to perform some proxying tasks until the failed node is back online. This can be done by looking at the routing table entries or by periodically polling resources. We do not do failure detection in our implementation.

5.4.1.d Startup Detection

All stored state directory entries must be restored on the corresponding nodes as early as possible in order to avoid the client to miss expected notifications and/or avoid erratic operation of the IoT application. Startup detection triggers restoration of the dynamic states in the node. If nodes proactively register their presence at boot time, this message can be used by the gateway as a mechanism to detect startup. In cases where the node does not get registered at the gateway proactively, other startup detection mechanisms should be in place. One possible method is observing node entries in the LLN routing table at the RPL root coupled with follow-up requests sent to the node. Any change in the LLN is reflected at the RPL root. When a node is not reachable for some time, this information may or may not reach the RPL root immediately. When it comes back, the node tries to become part of the DODAG again and hence the root tries to install this route in the routing table. This information will be communicated to the gateway to indicate that a node has attempted to rejoin the network. However, this does not necessarily mean

that the node is recovering from a failure. The node could have been unreachable for various reasons (e.g. mobility). The gateway may send a request to the client to check if an already stored state is available. If the requested dynamic state does not exist or if the value is different, the node is rebooting.

5.4.1.e State Restoration

The final step of the state recovery procedure is state restoration. Once the gateway realizes a node has just finished booting up, it starts the state restoration in a transparent way by replaying the original requests that resulted in the existing dynamic states. For instance, if a PUT request has set the threshold value of a resource, the same request will be sent to the node to restore this value. Similarly, an observe relationship between a device and the node will be restored by sending the same observation request by spoofing the IP address of the originating device. The same works for binding relationships. Recovering dynamic RESTlets may work in either of two ways. If the node has a permanent memory where the dynamic module is stored, the restoration process only takes the filename from the gateway and just does relocation and loading of the module from its local store. Otherwise, the gateway may replay the whole block-wise transfer request to transfer the dynamic modules from the SD.

5.4.2 Transparent Dynamic State Recovery for Encrypted Communication

Secured communication with constrained networks is difficult to be intercepted. In addition, a gateway cannot simply spoof and replay packets. However, [5.12], gives an innovative way of providing DTLS-based secured communication between LLNs and the Internet by introducing a trusted gateway. The solution exposes a virtual device for every physical device in the LLN at the gateway. This virtual device will minimally expose the same CoAP resources as the physical device. Security between the physical resource and a client in the Internet, is then provided by dividing the connection in two separate secured communication components. The first component securely connects the virtual resources at the gateway and the external client while the second component connects the virtual resource to the physical resource. This way, external networks only see virtualized devices that are perceived as real devices. Every packet addressed to such a virtual node will be terminated at the gateway and a separate packet is sent to the physical node. Responses are also handled in the same manner. This way, it becomes possible to intercept and inspect all packets, even when using DTLS. Of course, the underlying assumption is that the gateway is a trusted device. This can be compared to e.g. SSL/TLS termination in data centers for purposes of load balancing and deep packet inspection.

With this approach, transparent dynamic state recovery can be done, as the interception and inspection of packets takes place at the gateway after decrypting the packets coming from either side of the gateway. Once the inspection is done, the SD entries will be updated or new entries will be created as per the content of the packet. During recovery, the replayed packets from the SD will be encrypted before they are sent to the constrained node.

This mechanism will effectively address the recovery of dynamic states on constrained nodes without affecting the security of the whole system.

5.4.3 Other Dynamic State Recovery Mechanisms

5.4.3.a SD on External Storage

An alternative way of resuming communication after a node comes back from failure is to store a copy of all important dynamic states in external memory (Figure 5-9). Every time a node boots, it checks its own state directory for the availability of stored states. If there are stored dynamic states, it restores them back into volatile memory before continuing its normal operation. Once normal operation resumes, every change will be stored both in the volatile memory (RAM) and in the storage. In this approach, startup detection is an integral part of the firmware while failure detection is not applicable. State information collection is merely storing new states and modifications in the storage space as a copy. Recovery is the reverse process of copying information from external flash to memory. One of the advantages of this approach is that it does not require additional devices to collect and recover dynamic states. Because of that, no additional packet transmission is required. It is also the fastest way to recover the states. However, it has also its own limitations. First, this approach only works for nodes with external storage which is optional in many smart objects. As external devices are not aware of the process, it is not possible to keep connections alive temporarily in cases where the power cycling process takes longer. Finally, it only works for devices that have been designed in such a way and cannot be retrofitted to legacy devices.

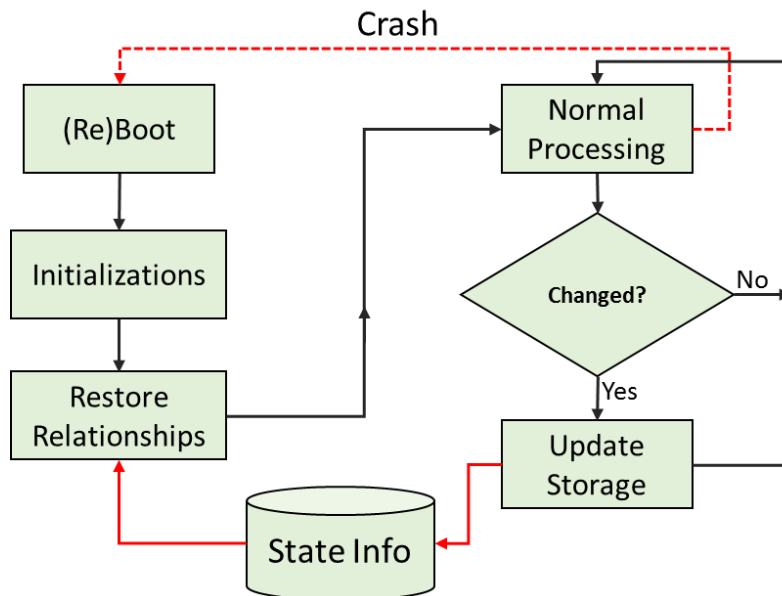


Figure 5-9: Dynamic State Recovery with State Directory on the Node

5.4.3.b SD on Immediate Parent in Multi-Hop Network

Another alternative is to put the state directory at the immediate parent of the node that operates in a multi-hop network. In tree-based multi-hop networks, all communication of a node with the external world goes through a parent node. This makes it easier for the node to store all dynamic states locally to make it available in case of failure of the child node. State information collection may take place transparently by intercepting every packet that passes through the parent node. The parent and child nodes periodically exchange control messages in order to indicate that they are alive. The parent is able to detect the failure of the child node when such messages do not arrive when expected. Upon detecting the failure, the parent node may take actions such as responding for requests on behalf of the child in order to defer re-registration or cancelation of pre-established relationships until the node is back online. The routing control messages that will be sent out upon boot time by a node also reach the immediate parent making its availability known to the parent. The parent may initiate the restoration process as soon as it receives such control messages. Since a third-party is involved in the process, state restoration takes place transparently without the knowledge of both the client and the sensor. In addition, the packet interception at the parent avoids additional packets that might be needed for state collection if the SD was placed elsewhere. Moreover, the parent works on behalf of the sensor until it is back online serving clients with strict deadlines. However, this approach puts a lot of load on the parent

node, which is most likely a constrained node itself. In addition, being a constrained device, the parent node may also fail losing all the stored information. Since there is no backup of the parent node, there is no way of putting the states back in the parent node when it resumes functioning after failure. Moreover, topology changes may result in changing parents which actually means losing the stored information.

5.5 Implementation

The transparent dynamic state recovery mechanism that has been presented in section 5.1.1 was implemented at the gateway node running CoAP++, an in-house implementation of the CoAP protocol and many additional features using Click Router [5.13]. The LLN devices are Zolertia Z1 motes [5.14] running Contiki 2.7 [5.15] and Erbium [5.16].

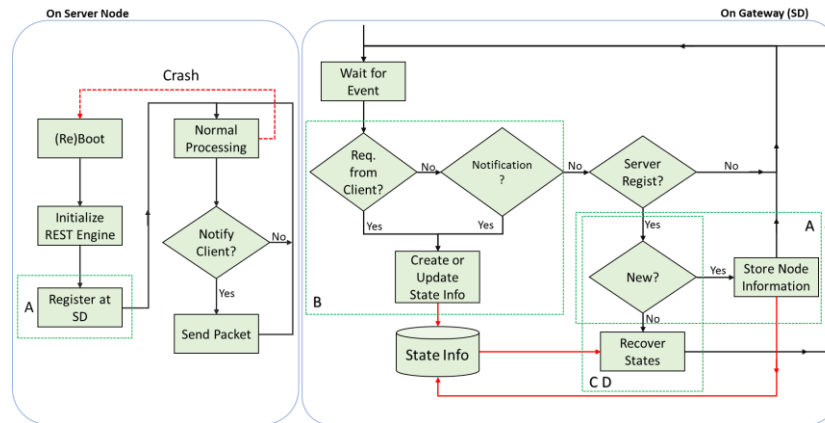


Figure 5-10: Implementation of Transparent Dynamic State Recovery

The SD functionality at the gateway performs the interception just after the IPv6 routing is done. After this step, the destination (sub) network is determined indicating if the transaction is from the LLN to an external network or vice versa. By doing the interception here, we can perform the required processing tasks depending on the origin of the packet. Packets originating from the LLN will be checked for notifications and the exact state information to store (or remove). Figure 5-10 shows the flow of control to successfully recover lost dynamic states of constrained nodes.

- A. SD – Node Association** – We make use of the proactive registration method by nodes to create the association between a node and the gateway. Since the node has earlier knowledge of the gateway, it will just send a registration request to the gateway after initializing its REST engine. Upon receipt of the request, the gateway checks if it has already stored information for that node.

The presence of such information ensures that the node is rebooting and triggers recovery. Otherwise, the node's information will be stored in the SD. We use confirmable blocking requests to block all other activities from commencing before the acknowledgement is received from the gateway. This way we can be sure that the association is created properly.

- B. **Dynamic State Collection** – dynamic state collection takes place by intercepting all traffic that comes from both sides – the LLN and external network. If the packet is originating from the external network, the state directory will be updated either by creating a new entry or by modifying existing one or even removing entries. Packets originating from the LLN always either modify the existing data (if they are notifications to observers) or remove the entry (if they are retransmissions and max-retransmission is reached).
- C. **Startup-Detection** – as mentioned above, a proactive registration request sent from an LLN node indicates that the node is booting and also alerts the gateway if there needs to be recovery attempts.
- D. **Dynamic State Recovery/Restoration** – the decision to restore dynamic states is made by the gateway after it gets a registration request from an LLN node and verifying that the node already has SD entries associated with it. If the stored information is an observe relationship or a PUT request, the gateway replays the original requests by using the original sender's IPv6 address as source address and using the values stored in the SD as required. For instance, the gateway uses the latest observe counter value and the observers IPv6 address when reestablishing the observation relationship and suppresses the response. Restoration of binding relationship requires specifying the binding information along with the observe option value set to 0. The gateway uses one of its own IPv6 addresses as the source address. Finally, to recover dynamic RESTlets, we just specify the name of the file in URI query while sending the recovery information. Upon reception of this packet the node looks for the file in its external memory and reloads it to memory.

5.6 Evaluation

5.6.1 Functional Evaluation

5.6.1.a Node-SD Association

```
[1] Packet to MyIP (cccc::1) (on port 0)
=====
Server Registration Request from [aaaa::c30c:0:0:2]. Recover Existing State Information.
SM Check and Start Recovery _____

Search Server
-----
State Information not found
-----
End Search Server
SM End Recovery
```

Figure 5-11: Registration of a node at the gateway

As explained before, at startup every node sends a CoAP request to a specific resource on the gateway to inform its availability. The message is sent as a blocking request so that every CoAP related operation is blocked until confirmation is received from the gateway. This is required in order to allow the Node-SD association to be in place before further interactions. The SD receives the packet and checks for stored dynamic states for that node and initiates recovery if found (Figure 5-11). Otherwise, the node is stored as a potential host for dynamic states.

5.6.1.b Dynamic State Collection

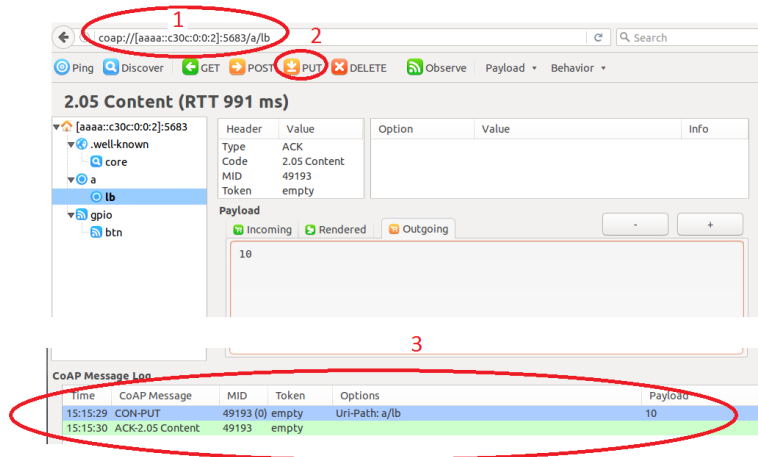


Figure 5-12: Dynamic State Collection (using PUT method)

Dynamic state collection is done by intercepting all traffic in a way that is transparent to both the client and the server. Figure 5-12 shows the CoAP Copper

Plugin setting the value of the `/a/lb` resource to 10 on the node with address `[aaaa::c30c:0:0:2]` through a PUT request (label 1 and 2 on the figure). The server responds that the operation was successful as indicated in label 3. This operation is totally transparent and both the client and the server are unaware of the interception made by the SD. When this request passes through the gateway, the SD intercepts the packet and stores the information locally as shown in Figure 5-13. The SD entry shows the client address, the server address and the entry type, among other values. Entry Type (ET) 2 means the record is for a PUT request.

```
[2] Start Intercept from Internet (on port 0)
=====
-->Intercepting Started<0> SRC<cccc::3> DST=<aaaa::c30c:0:0:2>
-----
Processing Intercepted Pkt
DST<aaaa::c30c:0:0:2> SRC<cccc::3>
-----
Start Extraction
-----
Girum: uriPath =<a/lb> entripath - <a/lb>
SD Extracted Values from Internet: Type [2] Client<cccc::3> Server <aaaa::c30c:0:0:2> url=<a/lb> MId = [49193]
ObsValue = <0> MsgID = <49193> Token[0]=<> ret<54144>
-----
End Extraction
First Actuation Request Received, update State Directory
-- Entries in State Directory--
SD: Entry [0] = <C = [cccc::3:37692] S = [aaaa::c30c:0:0:2] Obs=0 ET <2>>
-- Entries in State Directory--
-----
End Processing Intercepted Pkt
-----
-->Intercepting Ended<0> SRC<cccc::3> DST=<aaaa::c30c:0:0:2>
=====
[2] End Intercept (on port 0)
=====
```

Figure 5-13: Intercepted PUT Request from Client

5.6.1.c Observation Request and Notification Handling

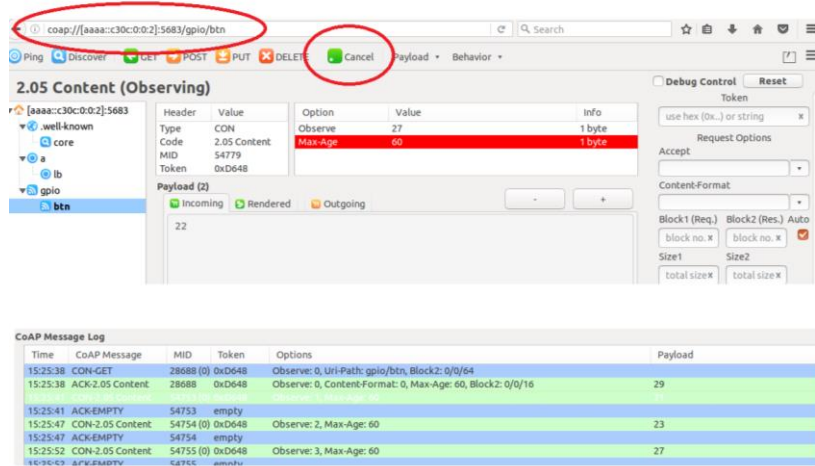


Figure 5-14: CoAP Observe Request Sent from Copper Client

As explained earlier, observation requests are treated differently as opposed to other intercepted requests. Figure 5-14 shows Copper sending an observation request to the `/gpio/btn` resource on node `[aaaa::c30c:0:0:2]`. Since this is the first observation request from this client, the observe counter is set to 0. This means, the SD needs to store an entry for this request after successful interception (Figure 5-15).

```

=====
[2] Start Intercept from Internet (on port 0)
=====
-->Intercepting Started<0> SRC<cccc::3> DST<aaaa::c30c:0:0:2>
=====
Processing Intercepted Pkt
DST<aaaa::c30c:0:0:2> SRC<cccc::3>

Start Extraction
=====
Run: urlPath =<gpio/btn> entypath - <gpio/btn>
SD: Extracted Values from Internet: Type [5] Client<cccc::3> Server <aaaa::c30c:0:0:2> url=<gpio/btn> Mid = [28688]
      ObsValue = <0> MsgID = <28688> Token[2]=<0H> ret<0>
=====
End Extraction
SD: Process Intercepted Pkt from Internet to: Server<aaaa::c30c:0:0:2> URI <gpio/btn>
SD: New Obs relationship from client cccc::3 to server <aaaa::c30c:0:0:2>/gpio/btn.
=====
Start Search Entry
++++Entry SD From Internet: Type = 5 Client cccc::3:38478 Server aaaa::c30c:0:0:2:5683/gpio/btn
SD: Stored Client <[cccc::3]:38478> Server <[aaaa::c30c:0:0:2]:5683/gpio/btn> obs <0> mid <28688> ret <0>
---Entries in State Directory---
SD: Entry [0] = <C = [cccc::3:38478] S = [aaaa::c30c:0:0:2] Obs=0 ET <5>>
---Entries in State Directory---

End Processing Intercepted Pkt
=====
-->Intercepting Ended<0> SRC<cccc::3> DST<aaaa::c30c:0:0:2>
=====
[2] End Intercept (on port 0)
=====

```

Figure 5-15: Interception of a New Observe Request at the SD

All notifications sent from the sensor node are intercepted and the SD Entry is updated. When the client wishes to stop the relationship, the RST message will be sent and the entry is removed from the SD.

5.6.1.d Dynamic State Restoration

```

=====
[19] Start Intercept from Internet (on port 0)
=====
-->Intercepting Started<0> SRC<cccc::3> DST<aaaa::c30c:0:0:2>
=====
Processing Intercepted Pkt
DST<aaaa::c30c:0:0:2> SRC<cccc::3>

Start Extraction
=====
Run: urlPath =<gpio/btn> entypath - <gpio/btn>
SD: Extracted Values from Internet: Type [5] Client<cccc::3> Server <aaaa::c30c:0:0:2> url=<gpio/btn> Mid = [55541]
      ObsValue = <0> MsgID = <55541> Token[2]=<0> ret<0>
=====
End Extraction
SD: Process Intercepted Pkt from Internet to: Server<aaaa::c30c:0:0:2> URI <gpio/btn>
SD: New Obs relationship from client cccc::3 to server <aaaa::c30c:0:0:2>/gpio/btn.
=====
Start Search Entry
++++Stored SD: Type = 2 Client cccc::3:58824 Server aaaa::c30c:0:0:2:5683/a/lb
++++Stored SD: Type = 2 Client cccc::3:33513 Server aaaa::c30c:0:0:2:5683/a/lb
SD: Stored client <[cccc::3]:52808> Server <[aaaa::c30c:0:0:2]:5683/gpio/btn> obs <0> mid <55541> ret <0>
---Entries in State Directory---
SD: Entry [0] = <C = [cccc::3:58824] S = [aaaa::c30c:0:0:2] Obs=0 ET <2>>
SD: Entry [1] = <C = [cccc::3:33513] S = [aaaa::c30c:0:0:2] Obs=0 ET <2>>
SD: Entry [2] = <C = [cccc::3:52808] S = [aaaa::c30c:0:0:2] Obs=0 ET <5>>
---Entries in State Directory---

End Processing Intercepted Pkt
=====
-->Intercepting Ended<0> SRC<cccc::3> DST<aaaa::c30c:0:0:2>
=====

```

Figure 5-16: List of Entries in the SD Before Node [aaaa::c30c:0:0:2]

The restoration process starts as the gateway receives a registration request from a node in the LLN. If entries exist in the SD, the SD initiates the restoration process and generates packets containing the right information to regenerate the dynamic states on the nodes. Figure 5-16 shows 3 entries in the SD before node [aaaa::c30c:0:0:2] reboots. There are two PUT requests and one Observe request stored in the SD for that node.

```

=====
[30] Packet to MyIP (cccc::1) (on port 0)
=====
Server Registration Request from [aaaa::c30c:0:0:2]. Recover Existing State Information.
SM Check and Start Recovery
-----
Search Server
-----
Found: EntryType = <2>
Client<[cccc::3]:50824>
Server<[aaaa::c30c:0:0:2]:5683>
Uri path <a/lb>
Token[0]= <4294935248>
Observe = <0>
Found: EntryType = <2>
Client<[cccc::3]:33513>
Server<[aaaa::c30c:0:0:2]:5683>
Uri path <a/m>
Token[0]= <4294935248>
Observe = <0>
Found: EntryType = <5>
Client<[cccc::3]:52808>
Server<[aaaa::c30c:0:0:2]:5683>
Uri path <gpio/btn>
Token[2]= <4294935248>
Observe = <10>
-----
End Search Server
+++++
Recover State Info
+++++

```

Figure 5-17: SD Containing 3 Records

Upon reception of a registration request from the node, the SD checks the list of entries associated with the registered nodes. Figure 5-17 shows the 3 records in the SD associated with the rebooted node. The first two records are PUT requests to resources `a/lb` and `a/m`, respectively, while the last record is an observation relationship. Accordingly, the SD initiates regeneration of the dynamic states on the node by sending the PUT and Observe requests to the node.

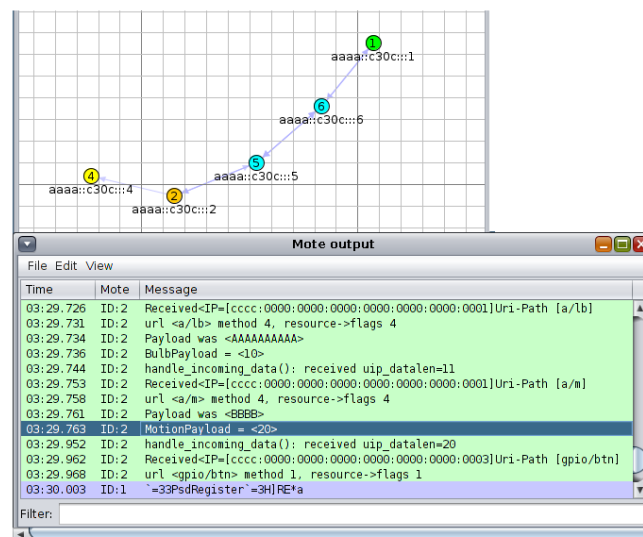


Figure 5-18: Output of the node after receiving the 3 packets from the SD

Figure 5-18 shows the outputs of the node on Cooja, illustrating the receipt of the 3 packets while Figure 5-19 shows the CoAP Message Log in Copper. When we closely look at the output of Figure 5-18, we see that the sender's address used to regenerate the observation relationship is the address of the client itself, i.e., [cccc::3]. This is important because the server always registers the client as an observer and sends notifications to him. Resumption of normal observation operation can be seen on Figure 5-19 by simply looking at the message IDs (MID). The change of MID from 12855 to 62713 indicates the re-initialization of the message IDs after reboot.

Time	CoAP Message	MID	Token	Options	Payload
15:57:53	ACK-EMPTY	12854	empty		
15:57:58	CON-2.05 Content	12855 (0)	0x0B2A	Observe: 10, Max-Age: 60	24
15:57:58	ACK-EMPTY	12855	empty		
15:58:42	NON-2.05 Content	62713	0x0B2A	Observe: 10, Content-Format: 0, Max-Age: 60	21
15:58:47	CON-2.05 Content	62714 (0)	0x0B2A	Observe: 11, Max-Age: 60	23
15:58:47	ACK-EMPTY	62714	empty		
15:58:52	CON-2.05 Content	62715 (0)	0x0B2A	Observe: 12, Max-Age: 60	27
15:58:52	ACK-EMPTY	62715	empty		

Figure 5-19: CoAP Message on Copper Showing Resumption of the Observe Operation

5.6.2 Performance Evaluation

5.6.2.a Delay Introduced by SD-Node Association

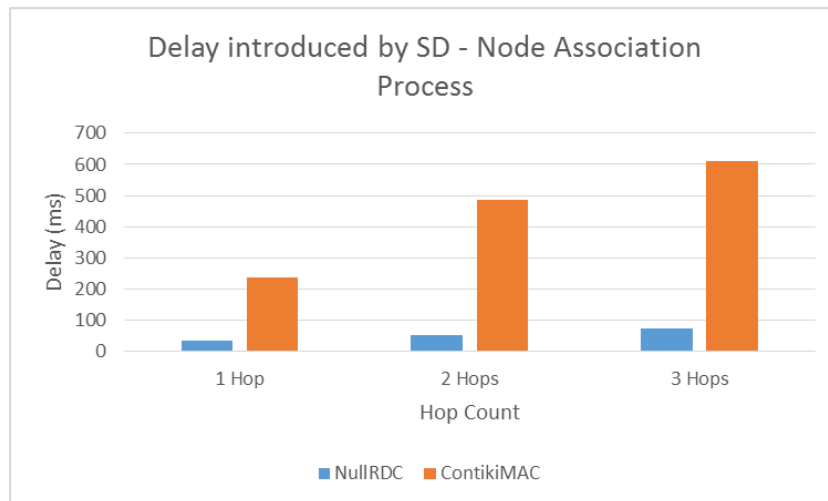


Figure 5-20: Delay Introduced by SD-Node Association Process

The first step in the transparent crash recovery solution is the association between nodes and the state directory. After rebooting, the node sends a confirmable blocking request to the SD in order to get registered. The SD registers the node and sends a confirmation back. This process introduces delay in the overall crash recovery process. Figure 5-20 shows the delay introduced due to this process using NullRDC and ContikiMAC as Radio Duty Cycling (RDC) protocols. In both cases, the delay increases with the hop count. This means that larger networks may suffer from longer delays. Yet, the delay for 3 hops is still less than 1 second for ContikiMAC and less than 100ms for NullRDC. Moreover, the increment is linear and the delay may not be very significant unless the network is too big. However, when we compare the results of ContikiMAC and NullRDC, the difference is quite high even for the same hop count. This is due to the nature of the two RDC protocols. NullRDC keeps the radio on all the time in order to receive packets as soon as the sender attempts to transmit them. This makes all transmission and reception faster but keeping the radio on all the time wastes energy. But ContikiMAC keeps the radio off 99% of the time in order to save energy. Due to this fact, senders have to wait for some time until the sleeping nodes are available making the communication delays higher.

5.6.2.b Impact of Intercepting Packets

Every packet that traverses the gateway needs to be intercepted which may introduce delays in the overall recovery process. We measured the arrival time difference by enabling and disabling interception and found out that the impact is minimal (<100ms). This is due to the fact that the interception is being handled by a non-constrained device.

5.6.2.c Impact of Routing Hops on the Recovery Process

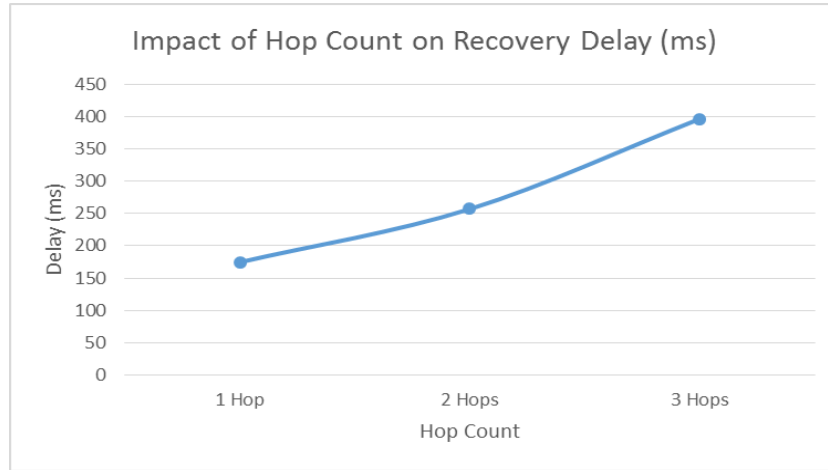


Figure 5-21: Impact of Hop Count on Recovery Delay

Delays are introduced when packets traverse from their source to destination. The delay is especially pronounced in LLNs. Therefore, it is important to study the impact of the distance, expressed in number of hops, between the node and the SD. As expected, the delay increases with the hop count (Figure 5-21).

5.6.2.d Impact of Number of Relationships on a Single Server

A single server may have multiple dynamic states created as a result of multiple requests from clients. When the node recovers from a crash, every dynamic state has to be recovered. In order to study the impact of multiple states on the recovery time, we sent 3 PUT requests to a single server and measured the recovery delay after the crash. We used ContikiMAC as Radio Duty Cycling (RDC) protocol which lets nodes sleep for most of the time to reduce energy consumed by passive listening. Figure 5-22 shows that the time required to recover states increases with the number of dynamic states required. The delay is due to the increased number of packets traversing the network, each containing information about a particular dynamic state that will be restored. The impact will be more visible when the number of hops increases because of the number of nodes it traverses to reach the server. When the number of states is a lot more than what we showed here, there is a possibility of congestion while trying to recover the states. In such cases, the SD must have a strategy to inject the recovery requests in the network.

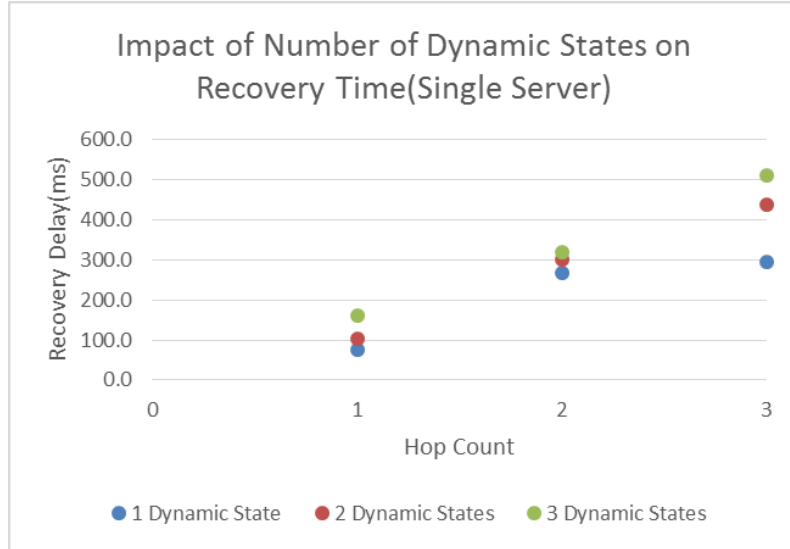


Figure 5-22: Impact of Number of Dynamic States that needs to be recovered on recovery time

5.7 Conclusion and Way Forward

CoAP-based IoT applications may depend on devices located inside a LLN. These devices may reboot for some reason or could be put offline temporarily for maintenance. Dynamic states, values which are created as a result of interaction with other nodes and that are stored in the volatile memory of the nodes, will be lost when the devices are back online. In this paper, we discussed the importance of dynamic state recovery in this context and the possible steps that might be taken to successfully recover the states. In addition, we presented a mechanism for the recovery of these dynamic states in a manner that is transparent to both the client and the server. Our proposed solution intercepts transactions with the potential of creating (and modifying) dynamic states on the nodes at the LLN gateway and stores relevant information in its state directory. The requests that are intercepted are PUT requests that may alter configuration parameters, observation requests, binding requests and dynamic deployment requests. The gateway is selected to host the state directory for two reasons. First, the gateway is a non-constrained device that can handle all interactions. Second, almost all such transactions originate from the outside network and go through the gateway to reach the LLN. The recovery process starts when a node reboots and sends a registration message to the gateway. Upon reception of the message, the gateway checks its state directory for any dynamic state information in the state directory. If a relevant directory entry exists, the gateway sends all the requests that created the dynamic

states to the node. The current work focuses on capturing interactions between clients residing outside the LLN and servers inside. In the future, we will work on recovery of dynamic states that are created due to interaction of nodes within the LLN.

Reference

- [5.1] Z. Shelby, K. Hartke, and C. Bormann, “RFC 7252: The Constrained Application Protocol (CoAP).” IETF, pp. 1–112, 2014.
- [5.2] K. Hartke, “RFC 7641: Observing Resources in the Constrained Application Protocol (CoAP).” IETF, pp. 1–30, 2015.
- [5.3] S. Cherrier, Y. M. Ghamri-doudane, S. Lohier, and G. Roussel, “Fault-recovery and Coherence in Internet of Things Choreographies,” pp. 532–537, 2014.
- [5.4] A. Akbari, A. Dana, A. Khademzadeh, and N. Beikmahdavi, “Fault Detection and Recovery in Wireless Sensor Network Using Clustering,” vol. 3, no. 1, pp. 130–138, 2011.
- [5.5] P. Milano *et al.*, “A Novel Technique for ZigBee Coordinator Failure Recovery and Its Impact on Timing A Novel Technique for ZigBee Coordinator Failure Recovery and Its Impact on Timing Synchronization,” no. November, 2016.
- [5.6] N. Tsiftes and A. Dunkels, “A Database in Every Sensor,” in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, 2011, pp. 316–332.
- [5.7] L. Selavo *et al.*, “LUSTER : Wireless Sensor Network for Environmental Research,” in *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007, pp. 103–116.
- [5.8] G. Teklemariam, J. Hoebeke, I. Moerman, and P. Demeester, “Facilitating the creation of IoT applications through conditional observations in CoAP,” *EURASIP J. Wirel. Commun. Netw.*, vol. 1, no. 1, 2013.
- [5.9] G. Teklemariam, F. Van den Abeele, I. Moerman, P. Demeester, and J. Hoebeke, “Bindings and RESTlets: A Novel Set of CoAP-Based Application Enablers to Build IoT Applications,” *Sensors (Basel)*, vol. 16, no. 8, 2016.
- [5.10] G. K. Teklemariam, F. Van Den Abeele, P. Ruckebusch, I. Moerman, and P. Demeester, “Dynamic Deployment of RESTlets on Constrained Devices (unpublished).”
- [5.11] P. Ruckebusch, E. De Poorter, C. Fortuna, and I. Moerman, “GITAR : Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules,” *Ad Hoc Networks*, vol. 0, pp. 1–25, 2015.
- [5.12] F. Van Den Abeele, T. Vandewinckele, J. Hoebeke, I. Moerman, and P. Demeester, “Secure communication in IP-based wireless sensor networks

via a trusted gateway Secure communication in IP-based wireless sensor networks via a trusted gateway,” no. October, 2015.

- [5.13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, “The Click Modular Router,” in *Proceedings of the 17th Symposium on Operating Systems Principles*, 1999, pp. 217–231.
- [5.14] Zolertia, “Z1 Datasheet,” pp. 1–20, 2010.
- [5.15] T. V. Adam Dunkels, Björn Gronvall, “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors,” in *29th Annual IEEE International Conference on Local Computer Networks*, 2004.
- [5.16] M. Kovatsch, S. Duquennoy, and A. Dunkels, “A low-power CoAP for Contiki,” *Proc. - 8th IEEE Int. Conf. Mob. Ad-hoc Sens. Syst. MASS 2011*, pp. 855–860, 2011.

6

Conclusion

“Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.”

—Antoine de Saint-Exupéry (1900 – 1944)

The advent of electromechanical technologies coupled with the evolution of wireless communication technologies created new possibilities and challenges to the Internet. Devices previously not considered to be online have now become an integral part of the Internet. Nowadays, household devices such as refrigerators and microwave ovens, medical utilities such as pacemakers are capable of communicating wirelessly. Novel applications involving self-driving cars, smart buildings and smart container tracking systems have appeared. All these innovations have something in common – smart objects. Smart Objects are sensors and actuators that have rather limited processing and communication capabilities. Despite this limitation, they are capable of sensing their environment and communicating the results after performing limited processing. They can also receive wireless signals from other devices and alter their environment.

Making use of these devices in a smaller scope with limited accessibility (e.g. inside a house) can easily be achieved by using proprietary protocols and devices. However, making these devices an integral part of the Internet is the biggest challenge for the current Internet. Firstly, the devices have severe resource constraints, making existing communication protocols impossible to be used. Secondly, several assumptions that are made while designing existing protocols may not hold for constrained devices due to their nature. Finally, there are new requirements of the constrained devices which are not addressed by existing protocols.

Several protocols have been developed to address the communication needs of constrained devices ranging from proprietary protocols designed to be used by devices manufactured by a specific company to open standards developed by international standardization bodies. In this PhD work, we focus on one of the standardized application layer protocols designed for constrained objects, named the *Constrained Application Protocol (CoAP)*. CoAP is a lightweight counterpart of the HTTP protocol. It uses the same GET, PUT, POST and DELETE methods for communication between clients and servers. One of the major differences between HTTP and CoAP is the fact that CoAP uses UDP at the transport layer while HTTP uses TCP. However, in order to achieve reliability CoAP uses Confirmable messages on top of UDP to let servers send Acknowledgements for every successful transaction. In order to allow seamless interaction between devices translation of CoAP into HTTP and vice versa is also possible. CoAP has several extensions that gives the protocol additional features. One of these extensions is Observe. The main goal of the PhD work is to leverage upon CoAP and design extensions in order to improve the performance of CoAP-based IoT applications.

6.1 Summary and Conclusion

In this dissertation, we presented a set of novel enablers that can make CoAP-based IoT applications more robust and flexible through extensions of the underlying protocols. One of the improvements discussed in this dissertation is the concept of conditional observation which is aimed at improving the standard CoAP observe functionality. The design of this concept was driven by the need to have a lightweight, efficient and compact solution for subscribing for very specific events, which is an important functionality when building IoT applications that directly interact with constrained devices. Normal observe in combination with client-side filtering can realize similar functionality, but suffers from the transmission of excessive packets that are not of interest to clients. We demonstrated the feasibility of implementing this functionality on constrained

devices. Using this implementation, we presented comparative results of using normal observation and client-side filtering versus conditional observation. We also presented theoretical evaluations of normal and conditional observation. From both the experimental and theoretical results, it is evident that conditional observations are a very useful extension to the basic observe behavior, both from an application point of view and from a network efficiency point of view. It enables clients to receive notifications that contain only state changes they are interested in. This has a twofold advantage: an application has the expressiveness to selectively collect data and the data of no interest does not have to travel over the network. The latter advantage will become even more important in larger constrained networks where notifications have to travel over multiple hops. As such, conditional observations can greatly contribute to the reduction of power consumption and increase of network lifetime. In addition, many scenarios can be thought of that can benefit from this functionality. As such, conditional observation is an interesting and easy-to-use enabler for many IoT applications.

The other enabler presented in this dissertation is the concept of Bindings. In monitoring applications that make use of the CoAP observe protocol, direct and flexible interactions between two constrained nodes (e.g., a sensor and an actuator) in a flexible way is not possible. The most common way used to curb this situation is for a non-constrained device (e.g., the LLN gateway) to establish an observe relationship with a sensor, get notified of every change and trigger an actuator based on the data received from the sensor. Our proposed solution introduces the concept of flexible bindings which are observation relationships between two devices established by a third party (e.g., smartphone). Once the relationship has been established, the initiator is no longer involved in subsequent communications between the two devices. The proposed solution, reduces the packet flow to the gateway and hence reduces the latency and the number of packets in the LLN compared to gateway or cloud based solutions. Through experiments we showed that the overhead (e.g., memory footprint) introduced by the binding solution is not significant compared to the gateway/cloud based solutions. In fact, regarding many aspects such as communication delay and number of packets, the binding solution outperforms traditional solutions. We also showed that this flexibility can be achieved by only making minor changes to the CoAP protocol and the observe extension.

RESTlets are the third novel enabler introduced in this dissertation. RESTlets are IoT application building blocks with data and control inputs, processing logic and data output. The RESTlet inputs can be associated with sensor or other RESTlet outputs through bindings to get input for processing. Once the RESTlet processes the data it produces outputs that will, in turn, be associated with other devices such as actuators or other resources in the LLN gateway or in the cloud. RESTlets can

be defined once and be instantiated multiple times. In this PhD work, we showed that by using RESTlets as IoT application building blocks, we can do in-network processing and aggregation in order to reduce the number of packets that traverse the whole LLN to the edge of the network and/or to the cloud which otherwise would lead to higher latency. We also showed that by interconnecting the data inputs and outputs of RESTlets to sensor outputs, actuator inputs or other RESTlets, we can easily build a complete IoT application within the LLN. Since the RESTlet approach allows distributed deployment of the processing logic at different nodes, there will not be too many resource hungry processes on one single node. It also gives greater flexibility in developing IoT applications by placing simple processing functionality inside the LLN and more complex one at the gateway or in the cloud. We ran several experiments in order to evaluate the performance of our solution by comparing it to traditional gateway-based or cloud solutions by using a different number of data generating nodes, data generating gap and TX/RX ratio. In all cases, our solution is capable of outperforming traditional solutions in terms of latency. Interestingly, the RESTlet solution provides a very good opportunity to use visual programming techniques to reduce the IoT application development to a set of drag-and-drop or point-and-click activities.

RESTlets can be hosted on both constrained and non-constrained devices alike. We looked at the dynamic deployment of RESTlets on constrained and non-constrained devices separately as the two devices have completely different capabilities. Due to the memory and processing power scarcity of constrained devices, we dynamically deploy only one RESTlet per node which can be instantiated only once but the architecture allows for more RESTlets to be deployed. On the other hand, non-constrained devices may have dynamic or static RESTlets and may be instantiated multiple times. Dynamic deployment on constrained devices involves the creation of a dynamic module and the transfer of the entire program code into external storage of the constrained node using a bulk transfer method such as CoAP Block-wise transfer. The dynamic loader of the node's firmware will then relocate the code in memory (RAM/ROM) to make it part of the firmware. The deployment process is completed with the creation of CoAP resources for all inputs, outputs and control parameters. This provides greater flexibility to the RESTlet concept by giving an IoT application developer the possibility to select and deploy RESTlets even after the nodes have been deployed in the field. However, this advantage comes with a tradeoff. The first tradeoff is the increased memory requirement due to the inclusion of the Contiki dynamic loader and the file system. But, despite the increase in memory footprint, we have shown that the entire firmware still fits in Class 0 constrained devices. The deployment time and energy consumed for dynamic deployment is another tradeoff. Evaluations performed on Z1 nodes show that a reasonable sized

dynamic module can be deployed in less than 10 seconds on a node after 3 hops (using ContikiMAC as RDC protocol). Such a deployment will consume energy, but, in turn, the data processing capabilities offered by the RESTlet may lead to a reduction of the number of packets transferred in the network and thus an overall reduction in energy consumption after some time.

CoAP-based IoT applications may depend on devices located inside a LLN. These devices may reboot for some reason or could be put offline temporarily for maintenance. Dynamic states, values which are created as a result of interaction with other nodes and that are stored in the volatile memory of the nodes, will be lost when the devices are back online. In this dissertation, we discussed the importance of dynamic state recovery in this context and the possible steps that might be taken to successfully recover the states. In addition, we presented a mechanism for the recovery of these dynamic states in a manner that is transparent to both the client and the server. Our proposed solution intercepts transactions with the potential of creating (and modifying) dynamic states on the nodes at the LLN gateway and stores relevant information in its state directory. The requests that are intercepted are PUT requests that may alter configuration parameters, observation requests, binding requests and dynamic deployment requests. The gateway is selected to host the state directory for two reasons. First, the gateway is a non-constrained device that can handle all interactions. Second, almost all such transactions originate from the outside network and go through the gateway to reach the LLN. The recovery process starts when a node reboots and sends a registration message to the gateway. Upon reception of the message, the gateway checks its state directory for any dynamic state information. If a relevant directory entry exists, the gateway sends all the requests that created the dynamic states to the node.

6.2 Future work

A number of novel ideas have been introduced and implemented in this PhD work. Yet, further enhancements and optimizations are possible to improve the performance of the solutions and can be interesting topics for future research work.

The current work on conditional observation assumed one conditional observation relationship on one resource. However, the impact of **multiple dissimilar conditional observation relationships** on a single resource needs further study. Moreover, the conditional observation solution can benefit from an efficient mechanism that **aggregates multiple relationships into a single one**. Further research is required to provide this solution.

From the tests conducted on the flexible bindings, we noticed that the relative position of the sensor and actuator results in varied performance in terms of latency and power consumption. The gain can be further optimized by performing **cross-layer optimization** techniques that look into the bindings and optimize the communication between the sensors and actuators that have binding relationships. For instance, ongoing research in 802.15.4e, particularly in IPv6 over the TSCH mode of IEEE 802.15.4e (6TiSCH) can be further investigated to find optimal performance using targeted scheduling. All in all, optimization of all communication protocols from the application layer to the physical layer should be investigated thoroughly in order to come to an optimal solution. Further work needs to be done also on finding a transparent solution to find out existing binding relationships and build a binding directory by using the concepts introduced in the crash recovery solution.

The RESTlets also benefit from such optimization as their performance is directly related to the binding relationships between components. In addition to this, the tests conducted showed that placing the RESTlets closer to the data generating nodes results in a better performance of the RESTlet solution. Providing **mathematical models** that will lead to optimal placement of RESTlets is another area that needs further investigation. The concept may further be **extended to be used in more powerful nodes** by leveraging on trending technologies such as OSGi and Spring. Also, the easy-of-use for creating IoT applications based on this concept can be enhanced by studying easy-to-use programming interfaces. By providing visual programming tools, we may reduce RESTlet-based IoT application development to a simple drag and drop operation.

Likewise, further work needs to be done on the crash recovery solution too. The behaviour of the solution when there are 10s, 100s or even 1000s of states are stored should be studied. Particularly, **how to replay packets to avoid congestion** needs further investigation. In some cases, nodes may fail and become unusable. A mechanism of **quickly replacing a failed node** is also another future work. The solution must be able to capture the data both in the volatile and non-volatile memory of the failed node.

As mentioned repeatedly, the entire PhD work is focused on CoAP-based IoT applications. However, it is interesting to see the performance of our solutions against other similar protocols used in the IoT arena. Hence, **comparing our CoAP-based solutions against other similar protocols (eg. MQTT)** is another potential research area.

Another area for future work is the **impact of these solutions on Low Power Wide Area Network (LPWAN) technologies**. Since most of our solutions are

aimed at decreasing power consumption through reduced traffic flow, we believe that devices that use such technologies (e.g. LoRaWAN devices) benefit more from our solution.

Further standardization of our solutions may also be considered as future work. There are current initiatives in the IETF to standardize dynamic resource linking for constrained RESTful environments using a different approach. We can look into possible integration of our solutions with the proposed internet drafts.

