FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Edge-to-Business Value Chain Delivery via Elastic Telemetry of Cyber-Physical Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Soheil Qanbari
Matrikelnummer 1129801

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar

Diese Dissertation haben begutachtet:

Prof. Dr. Schahram Dustdar   Prof. Dr. Frank Leymann

Wien, 2. Dezember 2015

Soheil Qanbari

Edge-to-Business Value Chain Delivery via Elastic Telemetry of Cyber-Physical Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Soheil Qanbari
Registration Number 1129801

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar

The dissertation has been reviewed by:

Prof. Dr. Schahram Dustdar	Prof. Dr. Frank Leymann

Vienna, 2nd December, 2015

Soheil Qanbari

Erklärung zur Verfassung der Arbeit

Soheil Qanbari
Schwemmgasse 2/3/55A, 1020 Wien, Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Dezember 2015

Soheil Qanbari

Danksagung

"*Betrachte den Menschen als ein Bergwerk, reich an Edelsteinen von unschätzbarem Wert. Nur die Erziehung kann bewirken, daß es seine Schätze enthüllt und die Menschheit daraus Nutzen zu ziehen vermag.*" Bahá'u'lláh

Die vorliegende Dissertation stellt eine Zusammenfassung all meiner akademischen Unternehmungen meiner Forschungslaufbahn in der Distributed Systems Group (DSG) der TU Wien dar. Sowohl die Höhen als auch die Tiefen meiner Forschungsarbeit haben mich wie ein pulsierender Herzschlag am Leben gehalten, um dieses Endergebnis zu erzielen. Diese Momente werde ich für den Rest meines Lebens in Ehren halten.

Meine aufrichtige Wertschätzung und Dankbarkeit möchte ich Prof. Schahram Dustdar ausdrücken, dessen wohlwollende Führung, großzügige Unterstützung und kontinuierliche Ermutigung diese Dissertation ermöglicht haben. Ich darf mich ganz und gar glücklich schätzen, ihn als Mentor zu haben. Ebenfalls möchte ich Prof. Hannes Werthner und Prof. Gerti Kappel danken, dass sie mich als Teilnehmer des Doctoral College of Adaptive Distributed Systems willkommen geheißen haben.

Weiters möchte ich all meinen Kollegen von der DSG für die fruchtbringenden Gespräche, die Unterstützung und Zusammenarbeit danken. Insbesondere, möchte ich Herrn Alexander Knoll für die aufrichtige Zusammenarbeit danken, der mir dabei half den Edge Cluster zu bauen.

Wahre Freunde erheben einen zu der wahren Stufe, die einem bestimmt ist! Ich bin dankbar für meine wundervolle Familie und wahren Freunde. Insbesondere danke ich Serwa Sabetghadam, Negar & Saied Khosravani, Alexandra & Puria Mahalli, Puneh & Michel Zarifzadeh, Matthew Stevens, Mirela Riveni, Rostyslav Zabolotnyi, Katrin & Victor Ciulian für ihre fürsorgliche Unterstützung und wahre Inspiration. Mein besonderer Dank gilt auch dem kleinen Sonnenschein, Lara Marie Ciulian, die ein Herz voller Liebe besitzt.

Meine äußerste, tiefempfundene Dankbarkeit gilt meiner Ehefrau, Samira, für ihre Liebe, Unterstützung und ihr Verständis. In der Tat war sie stets eine Quelle der Hoffnung. Ich erinnere mich an alle schlaflosen Nächte, in denen wir gemeinsam über die Herausforderungen in meiner Forschungsarbeit berieten. Unser Leitspruch lautet: Wir sollen glänzen!

Abschließend, möchte ich auch meinen Dank für die finanzielle Unterstützung des Vienna Doctoral College of Adaptive Distributed Systems (ADSys) ausdrücken.

Acknowledgements

"*Regard man as a mine rich in gems of inestimable value. Education can, alone, cause it to reveal its treasures, and enable mankind to benefit therefrom.*" Bahá'u'lláh

This PhD is the distillation of all my academic endeavors of my research career in the Distributed Systems Group at TU Wien. Both ups and downs of my research heartbeat kept me alive to achieve this ultimate work. I will cherish those moments for the rest of my life.

I would like to convey my sincere gratitude and thanks to Prof. Schahram Dustdar whose friendly guidance, generous support, and constant encouragement have made this thesis possible, and I am truly privileged to have him as my mentor. Likewise, I would like to thank Prof. Hannes Werthner and Prof. Gerti Kappel for welcoming me as a participant of the Vienna Doctoral College of Adaptive Distributed Systems.

Furthermore, I want to thank all my colleagues from the Distributed Systems Group for the fruitful discussions, support and collaboration. Notably, I appreciate Alexander Knoll for a genuine cooperation in helping me to build the edge cluster.

True friends elevate you to the true station destined for you! I appreciate my wonderful family and true friends, specially Serwa Sabetghadam, Negar & Saied Khosravani, Alexandra & Puria Mahalli, Puneh & Michel Zarifzadeh, Matthew Stevens, Mirela Riveni, Rostyslav Zabolotnyi, Katrin & Victor Ciulian for their caring support and sharing true inspiration. My very special thanks go to the little Miss sunshine, Lara Marie Ciulian who has a heart full of love.

My most heartfelt gratitude goes to my dear wife, Samira, for her love, support, and understanding. It is very true that she was always the source of hope. I still remember all the sleepless nights in which we consulted on research challenges together. Our motto of life is: *we shall shine*!

Finally, I am grateful for the financial support from the Vienna Doctoral College of Adaptive Distributed Systems (ADSys). The research leading to these results is sponsored by the Doctoral College of Adaptive Distributed Systems at the Vienna University of Technology.

Kurzfassung

Cyber-physischen Systeme (CPS) bewirken großartige Weiterentwicklungen im Bereich der IT und verteilten Computer Systeme. CPS verwandeln IT Randbereiche in lebendige Ökosysteme und schaffen damit neue Business Möglichkeiten. Diese Weiterentwicklungen bilden Chancen für neue Kooperationsnetzwerke zwischen verschiedenen Unternehmungen, Dienstleistungen und Dingen(IoT), die wir Edge-to-Business (E2B) Wertschöpfungskette nennen. Mit dem Aufstieg von E2B Entwicklungen ergeben sich neue Möglichkeiten für Ihre Geschäftsmodelle. Wie sie Ihre Businessprozesse in Zukunft steuern hängt maßgeblich von Ihren Geschäftsmodellen, Design Prinzipien, wie sie Ihre Anwendungen erstellen und welche Protokolle Sie zu deren Datenübertragung dafür nutzen, ab.

Zu diesem Zweck gibt es einen 3-Schritte-Plan: (i) Herauskristallisieren solcher neuen Möglichkeiten, (ii) Schaffen neuer robuster Design Modelle, (iii) Entwickeln von fairen flexiblem Messmechanismen zum erreichen und skalieren dieser sparsamen virtuellen Wertschöpfungsketten.

Im ersten Abschnitt haben wir speziell zuerst über die Entwicklungsmöglichkeiten und Modelle von CPS Anwendungen in den Bereichen für Cloud-Computing, Open Government Data und Health Care gesprochen. Um mehr ins Detail gehen zu können ist es nötig uns in Produkt Topologie Beispiele von Marktreifen Cloud Computing Services einzuarbeiten. Als nächstes verfassen wir eine originelle Abstraktion namens Gov. Data Compute Unit (DCU), sodass Organisationen in der Lage sind Entwicklern formale, strukturierte und programmierbare Daten Quellen Einheiten vielmehr Daten Kataloge in die Hand geben können. Zum Schluss entwickeln wir eine Lösung für die Semantik von Sensoren die Daten abfragen vom sogenannten Edge-Device-Layer. Das sollte die Qualität von unbearbeiteten Senior Daten steigern und sie besser interpretierbar machen.

Der zweite Part der These beschäftigt sich mit der Entwicklung von acht IoT Design Vorschlägen im Bereich der computerisierten Konstruktion von technischen Rand Anwendungen. Solche Abstrakten Lösungen befassen sich mit dem Grunddesign und den Grundprinzipien von solchen CPS Lebenszyklen. Unser Schema deckt folgende Anwendungsmöglichkeiten ab: Wiederholungen von zum Beispiel Bereitstellungszeiten, Begrenzte Anwendungsbereitstellung zugeschnitten auf die Datenleitung, Dynamische Anwendungskonfigurationen, Eingeschränkte Anwendungsbenachrichtigungen, Datenübermittlung im Gerät selbst und Wearable façade Entwicklungen, ab.

Um die Sache abzurunden gibt es ein eigenes Fernmessprotokoll das entwickelt wurde um als dritter Part die generierten Werte und Daten der CPS Anwendungen zu erfassen. Wir bieten ein Diameter of Thinks (DoT) Protokoll welches einem Werkzeug für Echtzeit-Messungen von Prepaid und Zahlen-nach-Bedarf Wirtschaftsmodell entspricht. Das DoT Protokoll behandelt zeitliche und aktionsabhängige Fernzugriffe.

Es ist nun an der Zeit sicherzustellen und zu überprüfen ob unsere Entwicklung Ihre Richtig umgesetzt wurde und einen nutzen für unsere Arbeit bringt. Die technischen Methoden in dieser Abschlussarbeit geben ein besseres Verständnis von die E2B Wertschöpfungskette und können beliebig erweitert werden auf mehr unterschiedliche Geschäftsmodelle, mehr Rand-Design-Modelle und mehr komplexe Messmodelle.

Abstract

The Cyber-Physical Systems (CPS) brought great computational advancements to transform the computing paradigm into a living ecosystem from edge infrastructures to business value propositions. Such advancements form a collaboration network among various entities, services and things which we call Edge-to-Business (E2B) value chain. However, along with the rise of E2B evolution, there is a need for blueprints on how to define your business models, design principles on how to build your applications, and a protocol on how to generate revenues from your ecosystem.

The three-part work plan of the this thesis is to: (i) discover such blueprints, (ii) provide robust design patterns; and (iii) develop fairly flexible metering mechanisms to achieve economies of scale within this virtual value chain.

In part one, we first identify the value creation models in the CPS application domains of cloud manufacturing, open government data and health-care. To be more specific, we incorporate product topology templates as a marketable entity for cloud manufacturing service. Next, we propose a novel abstraction unit called Gov. Data Compute Unit (DCU), so that governments are able to feed developers with formalized, structured and programmable data resource units rather than just data catalogs. For the last domain, we develop a solution for the semantic sensor data retrieval from the edge device layer. This enriches the quality of raw sensed data and makes it more interpretable.

The second part of the study deals with developing eight IoT design patterns as computational constructs for engineering edge applications. Such abstract solutions comprise core design principles through CPS life cycle. Our patterns cover applications' iteration cycles like provisioning, edge deployment pipeline, dynamic configurations, constrained application messaging, in-device data processing and wearable façade creation.

To complete the story, a telemetry protocol is developed at part three to capture the generated value of CPS applications. We offer the Diameter of Things (DoT) protocol that implements a real-time metering in the case of prepaid and pay-per-use economic models. The protocol handles time-based and event-based telemetry patterns.

Developed approaches and solutions were reasonably validated to prove the validity and utility of our work. The techniques developed in this thesis give a better understanding of the E2B value chain and can be extended to deliver more diverse business models, more edge design patterns as well as more complex metering models.

Contents

List of Figures

List of Tables

List of Listings

Prior Publications

The core of this dissertation is composed of a set of selected prior publications in scientific conferences, workshops, books and journals. The thesis consists of three subsequent parts keeping publication papers together in proper order. Please refer to Appendix 9.2 for a list of all publications of the author of this dissertation.

- Soheil Qanbari, Negar Behinaein, Rabee Rahim zadeh and Schahram Dustdar, Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications, *Internet Engineering Task Force (IETF).*
- Soheil Qanbari, Rabee Rahim zadeh, Negar Behinaein, and Schahram Dustdar, Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications, *12th International Conference on Economics of Grids, Clouds, Systems, and Service (GECON 2015)*, GECON-Conf, Cluj-Napoca, Romania, 15-17 September.
- Soheil Qanbari, Samim Pezeshki, Rozita Raisi, Samira Mahdizadeh, Rabee Rahim zadeh, Negar Behinaein, Fada Mahmoudi, Shiva Ayoubzadeh, Parham Fazlali, Keyvan Roshani, Azalia Yaghini, Mozhdeh Amiri, Ashkan Farivarmoheb, Arash Zamani, and Schahram Dustdar, IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications. *IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI 2016)*, April 4-8, Berlin, Germany. In review.
- Soheil Qanbari, Ashkan Farivarmoheb, Parham Fazlali, Samira Mahdizadeh and Schahram Dustdar, Telemetry for Elastic Data (TED): Middleware for MapReduce Job Metering and Rating, *The 9th IEEE International Conference on Big Data Science and Engineering (BigDataSE) (IEEE BigDataSE 2015)*, Helsinki, Finland, 20-22 August, 2015.
- Soheil Qanbari, Negar Behinaein, Rabee Rahimzadeh and Schahram Dustdar, Gatica: Linked Sensed Data Enrichment and Analytics Middleware for IoT Gateways, *IEEE International Conference on Future Internet of Things and Cloud (IEEE FiCloud 2015)*, August 24-26, 2015, Rome, Italy.
- Soheil Qanbari, Navid Rekabsaz and Schahram Dustdar, Open Government Data as a Service (GoDaaS): Big Data Platform for Mobile App Developers, *IEEE*

International Conference on Open and Big Data (IEEE OBD 2015), August 24-26, 2015, Rome, Italy.

- Soheil Qanbari, Samira Mahdi Zadeh, Soroush Vedaei and Schahram Dustdar. CloudMan: A Platform for Portable Cloud Manufacturing Services, *IEEE International Conference on BigData (IEEE BigData 2014)*, 27-30 Oct, 2014, Washington DC, United States.

- Soheil Qanbari, Fei Li, and Schahram Dustdar. *Toward Portable Cloud Manufacturing Services, IEEE Internet Computing*, vol. 18, no. 6, pp.NN-NN, 2014.

CHAPTER 1

Introduction

Gartner, Inc. forecasts[1] that 25 billion connected “Things”, will be operational in 2020. From an industry ranking perspective, they emphasize that utilities will position in the No. 1 spot because of investment in smart meters, manufacturing will be second, and government will be third by 2020.

Such systems, as we have already conducted some research on these top three upcoming IoT application trends, provide huge opportunities but considerable challenges for the stakeholders. The opportunities are increasing as devices are evolving to embed more computational resources and communications capabilities, enabling them to process information flows. This contributes to a built-in situational awareness as well as exposing smarter behavior. Challenges are posed by the limitations of edge devices as resource-constrained IoT compute units.

Edge computing devices are becoming more diverse and versatile with wide range of functionalities, more mobile and wearable, and potentially more elastic in terms of dynamic computational resource leveling. This has led to the rise of fully fledged edge infrastructure. Businesses will benefit from such edge evolution to seamlessly adjust their offerings to constantly changing requirements of their clients. This also contributes to businesses agility by enabling the IoT service providers to respond faster to the demanding needs of the markets. Firms can benefit from this as an enabler in developing adaptive business models built upon IoT delivery models like utilities, cloud manufacturing and e-Government.

The emergence of cyber-physical systems (CPS)[2] (also known as the Internet of Things) incorporates cloud computing[3] and embedded virtualization[4] mechanisms to expose environment data to analytic endpoints to drive context awareness. Such endpoints offer the ability to discover hidden patterns of mutually correlated variables and uncover actionable information within raw data for more utility. To achieve this, the CPS services should reside at the edge infrastructure and network as opposed to services hosted in cloud. This led to the rise of Fog computing[5] benefiting from the edge resource pooling, data stream processing, resulting in superior quality of service.

Ultimately such edge infrastructure will change the way to extract values from services and to monetize IoT applications. Hence, cloud computing is tightly coupled with IoT and acts as a front end and enabler to expose edge devices as IoT services. Cloud computing monetizes computing resources using the pay-per-use economic model with elasticity. The granularity of computing unit for monitoring, metering, and rating is a virtual machine (VM) in IaaS cloud. It represents a coarse-grained resource unit and is usually metered like an hourly rate for every VM instance per hour usage. This poses challenges of surcharge of jobs lasting less/more than an hour. Jobs lasting for 61 minutes will unfairly be charged for two hours. The VMs are utilizing shared resource pool for the hosted applications, making it hard to achieve optimal utilization of such resources. Consequently, the subscribers have to be charged for the unused resources[6]. Such considerations are very much dependent on how the associated business model is defined.

1.1 Motivation

Here, we leverage an established Business Model Framework[7] to describe the business models of IoT. The framework is illustrated in Figure 1.1.

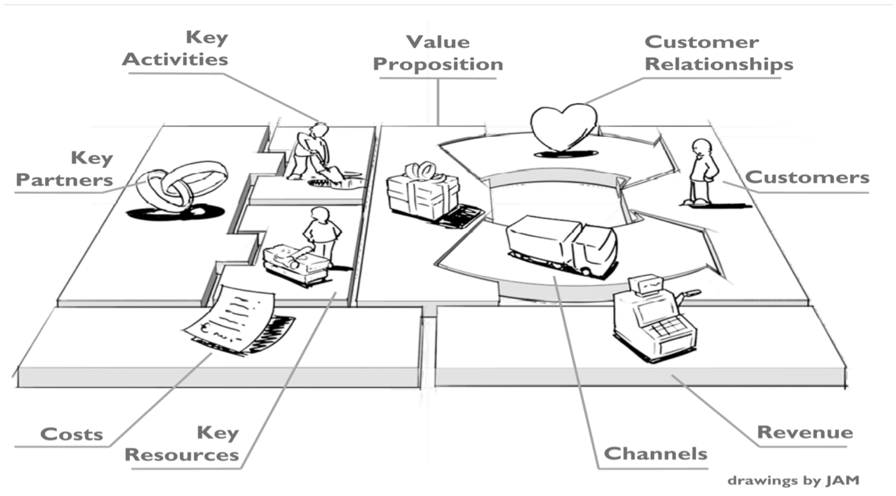


Figure 1.1: The method of describing business models

A business model is described in nine elements, which can be categorized in four areas—*Finance*, *Infrastructure*, *Customer* and *Value Proposition.* These four sections have strong and mutual interrelationships with one another that have to be taken into account in forming business models. Financial aspects aim at providing profitable and sustainable revenue streams. The cost structure in financial area is directly related to the stakeholders who are providing resources and conducting service activities, whereas the revenue structure is related to customers who are interested in the specific services. The monetary flow of this cost and revenue streams are effectively in use under the two models of "*Metered services* " and "*Subscription basis*". In the infrastructure area, the edge devices are provided with optimization capabilities. The infrastructure aspect offers embedded virtualization layer over edge underlying resources by providing utilization interfaces. The customer area's focus is on providing interfaces that define the consumer segments with their communication, distribution, and sales channels as touching point for service delivery. Overall, the three area converge on the value proposition of a business model. It seeks to solve customer problems and satisfy their needs with value propositions.

After defining the business model elements, we will be able to build "Virtual Verticals " as an instance of the business model. They are provided to domain-specific business services such as smart buildings or in our case, "Cloud Manufacturing (CMfg) ". It deals with configuring and deploying appropriate IoT services for distributed production and operational environment management in order to make the CMfg vertical application more efficient in its business and technological context. The "Vertical" means that such applications are delivered as an end-to-end service coverage including physical devices, middleware and applications for a certain physical environment. The "Virtual" part exposes the physical environment to an abstract unit like Virtual Factory for distributed manufacturing purposes.

Such deployed vertical applications posses diversity of edge sensors and actuators that will be creating streams of data. As the amount of data generated from edge services is immense and still growing, the data services business model takes a momentum here to address the data governance with a focus on data storage and processing aspects. Such data is big in terms of value, velocity, variety, veracity and potentially volume. Having the confidentiality of data properly managed, providing data to external experts or the public will further increase the utility of data. For instance, in case of smart cities, the government may expose the city data, which is rich in context, as services to civic-minded mobile developers for more utilization.

The IoT data service business model also should deal with data concerns like privacy enforcement, up-to-dateness, data availability and consistency in order to assure and improve data quality. In principle, the service provider does not have to own the data sources, nor does it provide applications. It only concerns management and provisioning of data. Both real-time data and persistent data can be in the scope of service.

Next is how to monetize such IoT vertical applications as well as their data assets. Although the pay-per-use economic model is pretty established in pricing cloud services, providers have to address a set of challenges when it comes to pricing edge services. IoT edge layer should be seen as constrained-node networks. The node is an edge device that host the IoT service. The hosting mechanism is basically fulfilled via container-based resource allocation. For instance, IoT services can be deployed on a Docker[1] as a

[1]https://www.docker.com

lightweight containerization technique. This elevates the need for finer-grained pricing models for IoT deployment units.

The challenges will increase when our edge infrastructure is composed of resource-constrained nodes or devices. The stringent constrained environment may limit our capabilities and enforce some restrictions, with unreliable communications, unpredictable bandwidth, and a highly dynamic topology[8]. To be more specific, the edge engineer should consider optimal resource control mechanisms to achieve an optimal utilization with regard to the following challenges stemming from the node constraints:

- Edge devices posses limited storage, footprint memory, and computing power to handle compute intensive processes;
- Due to cost constraints, the devices should function under low energy, limited battery and weak connectivity capabilities.
- Edge network topology is composed of constrained nodes which result in constrained network with low throughput/high packet loss.

1.2 Problem Statement

The aforementioned challenges above, establishes dependable and connected research areas: one focusing on driving novel business models from such constrained environment and its emitted data, and the next area on providing some design patterns as reusable computational constructs for designing, building edge applications for such constrained nodes. Last but not least, on how elastic telemetry of IoT applications can enable providers to achieve optimal utilization of their limited resources.

To formulate set of challenges that we take on and ultimately address, we outline the core problem statements. IoT service providers has to make architectural decisions on the three phases:

At the first step, they need to ensure that the essence of their business model innovation captures and generates values from the network of edge devices (e.g., sensors, actuators). In the definition of the business model, it is vital to clarify the granularity of the resource units that will be considered as a marketable entity. IoT application developers can compose and program such resource units with their utilization APIs, and expose them as services or mobile apps. Subsequently, the AppStore publishes these apps as marketable entities; then citizens, for instance, are able to subscribe, keep them in use and pay as they go. This leads to more economies of scale for all parties involved.

Once the first step is taken, we have the business model well defined and ready to implement. In fact, we go on to second step of developing the business models. There is a lack of unified standard for designing edge devices and gateways and this makes it hard to utilize IoT applications on so many diverse devices. Designing for an IoT is different since connected devices may use divergent types of networks and various connectivity patterns. To create a valuable, appealing, usable, and coherent edge application, we have to consider design on many different layers. This phase deals with providing design patterns that may leverage the performance and scalability of edge applications by giving adequate foresight to edge engineers.

Moving forward, we definitely want to take one step further to monetize our edge services. Such revenue generation of the utilized underlying resource is measured via metering and rating mechanisms. Telemetry is an evolving facet of utility computing that aims to leverage and expose computing resources as metered services and utilized assets. In particular, metering includes the processes of monitoring, aggregation, measuring and rating of an entire application, individual parts of an application, or tasks and underlying resources. Along with this idea, respecting resource capacity constraints on edge devices establishes a firm requirement for a mechanism in support of a telemetry of IoT applications. Such metering capability is needed when lack of resources among competing applications dictates our schedule and credit allocation.

Here, we need to implement a real-time metering mechanism of IoT services in the case of prepaid as well as pay-per-use economic models. It employs a mechanism to handle time-based and event-based telemetry patterns. The former is used for scenarios where the charged units are continuously consumed while the latter is typically used when units are implicit invocation events. The mechanism performs a metering transaction on edge services, collects the emitted usage data, then generates billable artifacts. Finally it permits micropayments to take place in parallel.

The main objective of this dissertation is to *enable fine-grained resource planning of IoT applications via elastic telemetry of edge services in support of adaptive business models for resource-constrained environments.*

With this objective in mind, our developed contributes to the following capabilities in:

1. defining adaptive business models for utilizing the IoT resource constrained compute units. Such business models can form a foundation for third party stakeholders to implement their own products on top and realize their own desired business transactions.
2. advising architects as well as developers with a set of established and unified design patterns to build robust and modular vertical applications.
3. revenue generation of IoT applications via telemetry services. Providers will be able to define their own complex metering schema of their applications and the subscribers will be transparently charged upon their granular resource and service usage respectively.

Our solutions, including developed models, middle-wares, design artifacts, and a protocol reveal promising resilience solutions for interested stakeholders. They will achieve a holistic view with an end-to-end coverage on designing, building and monetizing the IoT applications.

1.3 Research Questions

In this section, the contributions of this thesis are formulated in the following research questions. They are categorized in three main phases of the research and elaborated in a brief discussion.

Part 1) IoT Business Modeling

RQ 1: How does government become an open platform that allows civic-minded developers inside and outside government to develop practical applications? How to transform and expose government big data into tradable services?

This study addresses these challenges by driving new classes of GoDaaS business models, in which stakeholders participate and share things. These business models evolve through interactions between government and its citizens, like a service provider enabling its user community. The increasing amounts of government data made available, coupled with unpredictable and ever-changing business requirements, triggered us to incorporate cloud service delivery models to define flexible and adaptive business models. To this end, our contribution is threefold: (i) Deriving seven business models for Government Open Data as a Service (GoDaaS) platform. (ii) Novel government data abstraction unit, called Gov: Data Compute Unit(DCU) as a programming construct for developers. (iii) GoDaaS consultation service that generates a tailored and application specific business model.

RQ 2: How can we seamlessly achieve dynamic product topology configuration, automatic deployment and policy enforcement in a manufacturing as a service business model?

Cloud manufacturing is the convergence of utility-driven opportunities of novel distributed technologies, such as cloud computing and Internet of Things (IoT), coupled with collaborative engineering and applications that are transforming the manufacturing processes like supplement, consumption, and production models over the web. Thus, a cloud Manufacturing system is capable of serving multiple companies to planning, control and collaboration over the web. It has the capacity to make production planning and manufacturing work in progress (WIP) distributable, controllable, composable, configurable and portable. In this research question, we explore the solution in which such functionalities become feasible.

RQ 3: How pragmatic use of semantic web technologies can increase the quality of raw sensor data to make it more interpretable?

A data stream is a massive sequence of sensed data. Raw sensed data lacks semantics. This poses a challenge to apply analytics directly to raw IoT sensor data. Such operational data requires an intensive enrichment processes to drive value. It is a matter of real-time semantic sensor data retrieval. As such, the role of a semantic gateway which delivers automated and on-demand semantic annotations, labels and taxonomies for sensor data acquisition is vital. In this research study, we will illustrate how our solution transforms the sensor data to a linked data RDFs and then stream it to the analytics endpoints for querying and reasoning purposes.

Part 2) IoT Design Patterns

RQ 4: What kind of universal and abstract design principles and patterns are required to design, build and manage robust IoT applications?

Engineering edge applications requires abstracted design principles that prescribe how collaborating things should work. From the engineering perspective, a set of inter-related design patterns is required to realize all stages of the edge applications life cycle from their dynamic composition to their deployment and fulfillment. Such patterns can also become a unified language for system architects when designing their systems. In this research direction, we identify various computational constructs regardless of underlying technologies and details.

Part 3) Micro Telemetry

RQ 5: How can granular metering contribute to more utilization value for data-intensive applications? How does granular metering improve performance?

The quest for telemetry of the client's job resource usage becomes more challenging when the data-intensive job is deployed and processed in a distributed MapReduce model. Enriching MapReduce with telemetry services will contribute to more optimized resource utilization and performance in the cluster. The reason is granular elasticity control. Metering Map or Reduce tasks enables granular elasticity control on multiple levels by varying elasticity requirements like cost and quality. In this research work, we will examine how granular elasticity control is achieved via an optimal and tailored resource scheduling.

RQ 6: How can fine-grained telemetry of IoT deployment units respects resource capacity constraints on edge devices?

Respecting resource elasticity requirements on edge devices establishes a firm requirement of a lightweight protocol with support of fine-grained telemetry for IoT deployment units. Such metering capability is needed when lack of resources among competing applications dictates our schedule and credit allocation. In this research we identify two metering models of time-based and event-based patterns. The former is used for scenarios where the charged units are continuously consumed while the latter is typically used when units are implicit invocation events.

1.4 Scientific Contributions

This section reviews our scientific contributions in response to the identified and investigated research questions in Section 1.3. These contributions set a foundation for our research.

Part I) IoT Business Modeling

Contribution 1: Open Government Data as a Service (GoDaaS): Big Data Platform for Mobile App Developers.

The next web of open and linked data leverages governmental data openness to improve the quality of social services. This data is a national asset. In this study, we elaborate on this emerging open government movement, together with the underlying data transparency to drive novel business models which utilize these assets under a functioning platform called Open Government Data as a Service (GoDaaS). These business models actively engage civic-minded programmers in developing sustainable applications, contextualizing and utilizing the government open data resources. This leads to an expansive government marketplace, with many civic-minded developers might be new to doing business with the federal or state government. By means of a consultation service prototype, we provide development advices for programmers on how to work out the specific details of their applications business model. Having the business models in focus, this study also proposes a novel abstraction unit called Gov. Data Compute Unit (DCU), so that governments are able to feed developers with formalized, structured and programmable data resource units rather than just data catalogs. Such DCUs enable developers to cope with an increasing heterogeneity of state government data sets, by providing a unified interface on top of diverse data schemata from various states.

Contribution 2: Toward Portable Cloud Manufacturing Services.

In manufacturing, a product bill of materials (BOM) uses distributed manufacturing services for production purposes. Modeling BOMs poses challenges as regards distributed manufacturing plans, production policies, and the BOM's portability among multiple manufacturers. The authors' mechanism lets producers model and build BOMs by composing diverse manufacturing services and resources using the OASIS Topology and Orchestration Specification for Cloud Applications standard.

Contribution 3: CloudMan: A Platform for Portable Cloud Manufacturing Services.

Cloud manufacturing refers to "as a Service" production model that exploits an on-demand access to a distributed pool of diversified manufacturing services and resources.

It forms elastic and reconfigurable production lines, which enhance efficiency, by allowing optimal resource allocation in response to demand changes and market dynamics. This paper studies these challenges and proposes a portable cloud manufacturing platform, entitled “CloudMan” , aiming at achieving a portable deployment of cloud manufacturing services to any compliant distributed production line in the cloud. The stakeholders of CloudMan are detailed together with their API requirements, where each stakeholder has an interest. Having this rigorous analysis in mind, we present a holistic architecture for CloudMan, as it considers the manufacturing data, material and event flow from sensors and shop floors, through services to end products. In architecting such platform, there is a lack of agreed standard for the portability and orchestration of manufacturing services, as well as their definition. The proposed platform incorporates OASIS Topology and Orchestration Specification for cloud Applications (TOSCA) policies, plans and templates as a mechanism for dynamic configuration, portability and deployment of manufacturing services across multiple collaborating manufacturers. Thereby, the architecture provides a set of abstraction levels for various types of manufacturing services which encapsulates and addresses specific requirements to satisfy the needs of stakeholders.

Contribution 4: Gatica: Linked Sensed Data Enrichment and Analytics Middleware for IoT Gateways.

Raw sensed data lacks semantics. This poses a challenge to apply analytics directly to raw IoT sensor data. Such operational data requires an intensive enrichment processes to drive value. Pragmatic use of naming conventions and taxonomies can increase the quality of data and make it more interpretable. In this paper, we incorporate semantic and linked data technologies and offer a middleware called Gatica, to dynamically inject semantics to make the raw streaming data of an IoT gateway ”Rich” on the device layer. Gatica collects the real-time sensor data, enriches them using annotations, then transforms and exposes them in RDF triples, and finally streams RDF objects to the analytic endpoint for querying the linked sensor streaming data. Various analytic applications can utilize our middleware by sending SPARQL requests over the sensor network to our query interface and retrieving the results. Our middleware offers the ability to discover hidden patterns of mutually correlated variables and uncover actionable information within raw data for more utility. This paper details Gatica’s architecture together with its implementation.

Part II) IoT Design Patterns

Contribution 5: IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications.

The objective of design patterns is to make design robust and to abstract reusable solutions behind expressive interfaces, independent of a concrete platform. They are abstracted away from the complexity of underlying and enabling technologies. The connected things in IoT tend to be diverse in terms of supported protocols, communication methods and capabilities, computational power and storage. This motivates us to look for more cost-effective and less resource-intensive IoT microservice models. We have identified a wide range of design disciplines involved in creating IoT systems, which act as a seamless interface for collaborating heterogeneous things, and are suitable to be implemented on resource-constrained devices. The IoT patterns covered in this paper vary in their granularity and level of abstraction. They are inter-related, well-structured design artifacts, providing efficient and reliable solutions to recurring problems discovered by IoT system architects. The authors offer sound advice for designing, building, and scaling with cross device interactions inherent in complex IoT ecosystems.

Part III) Micro Telemetry

Contribution 6: Telemetry for Elastic Data (TED): Middleware for MapReduce Job Metering and Rating.

The consumption-based rating of MapReduce jobs is tightly coupled with metering the infrastructure resource usage it runs on. In this context, metering and controlling the job execution depends on the number and type of containers used to setup and run the Hadoop cluster as well as the duration of the job execution. Duration-basis metering like an hourly rate for every instance per hour usage, poses challenges of surcharge of jobs lasting less/more than an hour. Jobs lasting for 61 minutes will unfairly be charged for two hours. In response to these findings, the authors offer Job-basis telemetry mechanism rather than Duration-basis where the metering granularity is carried on MapReduce DAG bundles, jobs and tasks levels. This model is developed as an elastic data telemetry (TED) middleware to provide real-time resource utilization awareness over data intensive applications. Clients will benefit from this model by enforcing their applications elasticity policies and achieve pricing transparency over their actual usage. This granular elasticity control is achieved by moving jobs among priority queues which fit cost and quality requirements. TED collects the emitted usage data stream, generates billable artifacts to form a tailored policy (scale up/down) to satisfy several desirable properties. This contributes to a supervised, finer-grained resource allocation due to the application behavior.

Contribution 7: Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications.

The Diameter of Things (DoT) protocol is intended to provide a real-time metering framework for IoT applications in resource constrained gateways. Respecting resource capacity constraints on edge devices establishes a firm requirement for a lightweight protocol in support of fine-grained telemetry of IoT deployment units. Such metering capability is needed when lack of resources among competing applications dictates our schedule and credit allocation. In response to these findings, the authors offer the DoT protocol that can be incorporated to implement real-time metering of IoT services for prepaid subscribers. The DoT employs mechanisms to handle the composite application resource usage units consumed/charged against a single user balance. Such charging methods come in two models of duration-based and event-based patterns. The former is used for scenarios where the charged units are continuously consumed while the latter is typically used when units are implicit invocation events. The DoT-enabled platform performs a chained metering transaction on a graph of dependent IoT microservices, collects the emitted usage data, then generates billable artifacts from the chain of metering tokens. Finally, it permits micropayments to take place in parallel.

1.5 Structure of the Work

With this brief introduction in mind, the rest of thesis is organized as follows:

- Chapter 2 defines preliminary terms, concepts and fundamental information regarding the technologies used in the remainder of this thesis.
- Part I is focused on the scientific work related to the IoT business models. Three business models are explored in details and considered as the foundation for further research.

  ∘ Chapter 3 presents our scientific work on making business with the data. We address challenges like how governments can expose their data assets as services for more utilization. Here, a novel abstraction unit called Gov. Data Compute Unit (DCU) is proposed so that governments are able to feed developers with formalized, structured and programmable data resource units rather than just data catalogs.

  ∘ Chapter 4 extends the work on business modeling to the manufacturing world. We demonstrate how OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)[9] influence the cloud manufacturing services.

  ∘ Chapter 5 incorporates the semantic web technologies to enable semantic sensor data retrieval on the IoT device layer. This fulfills sensor data enrichment via pragmatic use of naming conventions and taxonomies.
- Part II consists of one chapter that defines eight IoT design patterns to build edge applications. The proposed patterns will guide the edge engineers and developers in building their edge applications. This middle part functions as a bridge between part one and part three.

  ∘ Chapter 6 covers the overall process to engineer IoT applications. The proposed patterns realize all stages of the edge applications life cycle from their dynamic composition, provisioning, deployment to their data processing, messaging and fulfillment.
- Part III consists of two chapters on enabling fine-grained telemetry of IoT applications. We will demonstrate how micro telemetry can contribute to more granular and transparent metering as well as better resource elasticity control in the constrained environment.

  ∘ Chapter 7 enriches MapReduce-based applications with telemetry services to achieve more optimized resource utilization and performance in the cluster.

  ∘ Chapter 8 defines a new protocol, called Diameter of Things (DoT) to meter IoT composite applications. The DoT contributes to fully implementing a real-time policy-based telemetry and resource control for a variety of edge services.

- Chapter 9 concludes and summarizes the content and purpose of the thesis. Looking forward, it provides a thoughtful outlook to our piece of work over future research directions.

CHAPTER 2

Background

In this chapter we present preliminary concepts, terms, standards, and technologies considered in the study. They form the basis for the work carried out in this thesis.

2.1 Cloud Computing

Even though the concept of *cloud computing* has more a business sense, it incorporates the virtualization technologies to elastically provision computing capabilities distributed over Internet [10] via utilization mechanisms [11].

The US National Institute of Standards and Technology (NIST) defines cloud computing as *"...a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [12].

Taking the NIST definition further, there are five characteristics that stand out. First is the *on-demand self-service* which enables service consumers to automatically provision computing resources in an on-demand fashion.

Second is *Broad network access*, which make the exposed capabilities available via a network for more utilization. Third deals with *Resource pooling.* The computing resources are pooled and shared among multiple consumers. The next important attribute is the *Rapid elasticity*, which is the core capability to cope with fluctuating load by dynamically provisioning and releasing cloud resources. We utilize this concept to enable elastic telemetry of applications. Last characteristic is the one which we focus on most. That is *Measured service.* We leverage this concept to the meter and measure various domains like metering data-intensive and CPS applications.

2.2 Cyber-physical Systems

Cyber-physical Systems (CPS)[13] are gaining a lot of attention of researchers and engineers as smart systems which utilizes the network of collaborating edge devices, computational components and human services. This contributes to enabling potential hybrid innovative applications. The NIST organization has recently released a draft framework[14] on the CPS. In addition to CPS, there exist many other terms like Industrial Internet, Internet of Things (IoT), Machine to Machine (M2M), smart cities that present similar systems and concepts. As such, approaches described in this this should be considered to be equally applicable to IoT and CPS.

In NIST CPS draft framework, some of the main characteristics of such systems are emphasized. First is the ability to consider CPS systems as a pervasive computing [15] environment. This confirms that such systems are embedding more computing resources within devices, enriching them with more connectivity and processing capabilities.

In architecting such Cyber-physical systems, the first layer is an edge device layer. Here, there is a network of devices, sensors, actuators which behave smartly due to the environment context. Such edge devices are the *things* in the Internet of Things (IoT). On top of this Cyber-physical device layer comes the software systems to virtualize and expose devices as services. Such services make the devices discoverable, addressable, configurable and controllable. These systems can be also considered as virtual verticals which can be diverse to various business domains like manufacturing, smart buildings and healthcare systems. The next layer will integrate all the system together and represent them as a System of Systems (SoS).

Last but not least, NIST defines the Cyber-physical systems as to integrate computation, communication, sensing, and actuation with physical systems to fulfill time-sensitive functions with varying degrees of interaction with the environment, including human interaction.

2.3 Business Modeling

A business model[16] thus describes the rationale of how an organization creates, delivers, and captures value, in economic, social, cultural or other contexts. For firms, business modeling is driven by the value chain as a sequence of exchanges of economic resources among agents - the process of delivering some resources to obtain more value. The business model concentrates on the economic sustainability of an entire resource exchange system (duality: give & take) among parties. We see three main elements to conceptualize business models: *Resources*, *Events* and *Agents*.

The cloud abstraction model delivers a shared pool of configurable computing resources (processors, storage, applications, etc.) that can be dynamically and automatically provisioned and released [3]. This elastic resource delivery of cloud computing enables business agility by enabling the providers to respond faster to the demanding needs of the markets. Firms can benefit from this as an enabler in developing adaptive business models built upon cloud delivery models. The cloud computing business model focuses on the value of business objects (cloud resources) exchanged among parties and abstracts away the technical and implementation details. Now we delve into the core concepts, their meanings and interdependencies:

◇ **Computing Resource** is a thing that has utility for the agents. In fact users need to plan, monitor, and utilize the resources. For instance, computing resources can be CPU, Storage, Memory, and VMs.
◇ **Agent** is an individual or a provider capable of having control over resources, and transferring or receiving the control to or from other individuals or organizations. Examples of agents are consumers, cloud providers, brokers.
◇ **Event** represents either an increment or a decrement in the value of economic resources. Events might demand or supply resources. Events can be classified into two poles of a duality pattern of *Give* and *Take.* At least one take event and one give event exist for each resource. When the event occurs, the provider loses rights to the resource, and the consumer receives the rights.
◇ **Commitment** is a promise or obligation of agents to perform an event affecting the resource.
◇ **Contract** is a collection of commitments and terms. Thus, the contract can specify what should happen if the commitments are not fulfilled.

2.4 Cloud Manufacturing

Cloud manufacturing (CMfg) [17] is a new manufacturing paradigm developed under the utilization of cloud computing, Internet of Things (IoT), virtualization technologies and collaboration engineering. he cloud computing paradigm provides a virtualization layer on top of shop-floor devices and resources envisaged in the IoT. These manufacturing resources can elastically scale with on-demand access among distributed regions, leading to an automated IoT application deployment. Collaborative engineering addresses the orchestration and collaboration side of manufacturing workflows, in which resources (such as devices, sensors, materials, and drivers) are virtualized and encapsulated into a product bill of materials. In cloud manufacturing, distributed resources are encapsulated into cloud services and managed in a centralized way [18].

In our study, a portable cloud manufacturing system serves multiple factories as an integrated manufacturing ecosystem, helping to govern the manufacturing process and support product design. It also enables overall dynamic configuration, production lifecycle management, and team collaboration. It makes production planning and manufacturing work in progress (WIP) distributable, controllable, composable, and portable over the Web, leading to an even broader definition for the concepts of “design anywhere and make anywhere (DAMA),” manufacture on demand, and manufacturing as a service.

2.5 Open Government Data

Governments are rich in valuable civil information, as they must re-imagine their roles as an information provider. The Gov 2.0 movement welcomes civic intelligence with its data transparency, i.e., enabling governments to adapt to the ever-changing needs of their citizens. The *Data.gov* initiative, for instance, does not just catalog raw data; it takes this idea to a new level by providing a collection of open APIs[1] to government data. The rationale for *data.gov* is laid out in a reliable information infrastructure that "*exposes*" the underlying data to the public (i.e., online) at large. The Gov 2.0 platform model takes the further step of highlighting third-party *mobile applications* created by independent developers in a real "*AppStore*" like Washington, D.C.[2]. These repositories of data-driven apps can be open-sourced as a means for sharing best practices with other governmental bodies and cities. *Code for America*[3] is an instance of such open source collaborative business model.

Opening up and publishing raw data such as maps, employment statistics, weather surveys, agricultural statistics, and educational records, together with their associated APIs, while enforcing and respecting privacy policies of course, makes two things possible: (i) Enabling government as a platform and data infrastructure provider, who leverages government platform to a new level of transparency to queryable sources of large amounts of underlying operational data. This leads to more control by citizens over their governments, as well as closer cooperation with them. (ii) Civic-minded programmers and the private sectors can build and deploy applications on government's data infrastructure to achieve optimal utilization of the data. This utilization is realized by creating new interfaces to government using the Open311[4] standard, developing *MobileApps* and offering new services in the *AppStore* as aided by government-provided data APIs. For instance, developers can register for a key at api.data.gov to access data offerings, via REST-full requests and returned responses in JSON or XML.

2.6 TOSCA

The Topology Orchestration Specification for Cloud Applications (TOSCA)[19] is an OASIS standard that introduces a grammar for describing service templates by means of *Topology Templates* and *Plans*. The focus is on design time aspects, i.e. the description of services to ensure their exchange. Runtime aspects are addressed by providing a container for specifying models of plans which support the management of instances of services. In fact the designers will be more focused on the service topology which is a logical relationship between services. The service topology defines the resource specification without any explicit implementation details. With this approach the implementation of a service can change without any effect to the cloud application design.

[1] http://data.gov/developers/apis
[2] http://Apps.DC.gov
[3] http://codeforamerica.org/
[4] http://open311.org

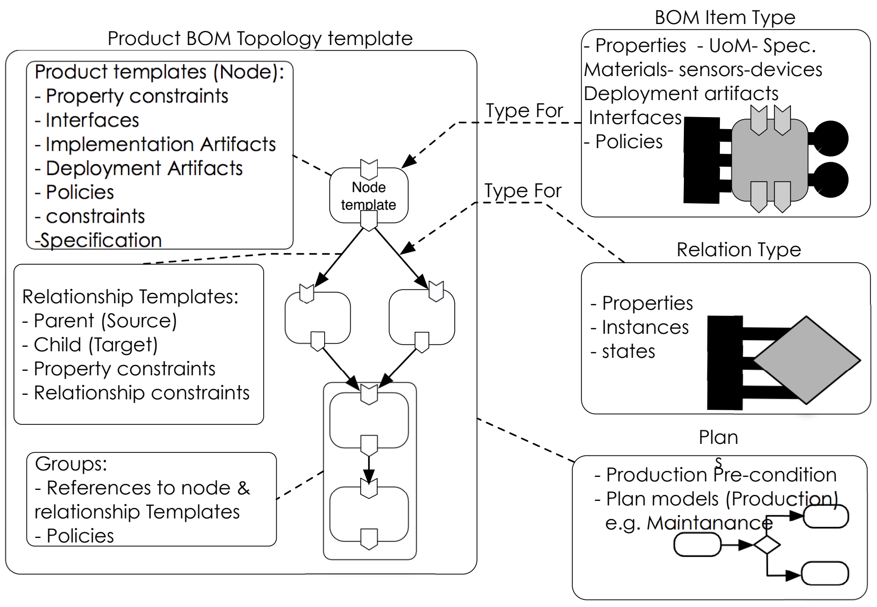


Figure 2.1: The core elements of TOSCA Service Template is the Service Topology.

Figure 2.1, shows the meta model of the TOSCA. The root of a TOSCA service is the *Service Template*. The service template contains a directed graph that represents the structure of the service called a Service Topology. Every service template has at least one *Service Topology*. The topology graph is composed of nodes and edges. Edges in a directed graph are links with a direction from node to node. The edges in a Service Topology graph are binary relationships between nodes. The nodes represent the logical components of the service. These nodes and relationships are templates that are patterns for the real nodes and relationships instantiated in a deployed service. Plans orchestrate various aspects of a service life cycle. The TOSCA specification defines build Plans and termination Plans. Build Plans orchestrate the deployment and installation of a service. Termination Plans orchestrate decommissioning a service. Designers of TOSCA services can add plan types as needed. The designers can also benefit by workflow notations such as BPMN or BPEL.

Part I

IoT Business Models

CHAPTER 3

Open Government Data

3.1 Introduction

Information theory begins with the representation, interpretation and transmission of patterns of data, i.e., patterns made up of different kinds of "*things*". We attach meanings to these patterns and call the result, "*information*". Patterns of data are mainly of interest when they are transmitted from a source to a receiver [20]. Along with this idea, governments are rich in such valuable information, as they must re-imagine their roles as an information provider. The Gov 2.0 movement welcomes civic intelligence with its data transparency, i.e., enabling governments to adapt to the ever-changing needs of their citizens. The *Data.gov* initiative, for instance, does not just catalog raw data; it takes this idea to a new level by providing a collection of open APIs[1] to government data. The rationale for *data.gov* is laid out in a reliable information infrastructure that "*exposes*" the underlying data to the public (i.e., online) at large. The Gov 2.0 platform model takes the further step of highlighting third-party *mobile applications* created by independent developers in a real "*AppStore*" like Washington, D.C.[2]. These repositories of data-driven apps can be open-sourced as a means for sharing best practices with other governmental bodies and cities. *Code for America*[3] is an instance of such open source collaborative business model.

Opening up and publishing raw data such as maps, employment statistics, weather surveys, agricultural statistics, and educational records, together with their associated APIs, while enforcing and respecting privacy policies of course, makes two things possible: (i) Enabling government as a platform and data infrastructure provider, who leverages GoDaaS to a new level of transparency to queryable sources of large amounts of underlying operational data. This leads to more control by citizens over their governments, as well

[1]http://data.gov/developers/apis
[2]http://Apps.DC.gov
[3]http://codeforamerica.org/

as closer cooperation with them. (ii) Civic-minded programmers and the private sectors can build and deploy applications on government's data infrastructure to achieve optimal utilization of the data. This utilization is realized by creating new interfaces to government using the Open311[4] standard, developing *MobileApps* and offering new services in the *AppStore* as aided by government-provided GoDaaS data APIs. For instance, developers can register for a key at api.data.gov to access data offerings, via REST-full requests and returned responses in JSON or XML.

Moving forward, we frame the research question of GoDaaS: How does government become an open platform that allows civic-minded developers inside and outside government to develop practical applications? How to transform and expose government big data into tradable services? This study addresses these challenges by driving new classes of GoDaaS business models, in which stakeholders participate and share things. These business models evolve through interactions between government and its citizens, like a service provider enabling its user community. The increasing amounts of government data made available, coupled with unpredictable and ever-changing business requirements, triggered us to incorporate cloud service delivery models to define flexible and adaptive business models. To this end, our contribution is threefold: (i) Deriving seven business models for Government Open Data as a Service (GoDaaS) platform. (ii) Novel government data abstraction unit, called Gov. DataComputeUnit (DCU) as a programming construct for developers. (iii) GoDaaS consultation service that generates a tailored and *application-specific* business model.

The chapter continues with a survey on some contemporary related work on defining open government business models at section 3.2. With some definitive clues on how the government data is currently traded, we define a new abstraction unit, called *Gov. Data Compute Unit (DCU)*, for government data at section 3.3. This unifies our data resource trading unit within all proposed business models. To elicit and illustrate the need for GoDaaS from the requirements engineering perspective, section 3.4 is devoted to the core stakeholders and their relationships in GoDaaS ecosystem. Having the stakeholders' interest in the provision of government as GoDaaS platform, the focus is put on implementing this new model with the API requirements at section 3.5. Subsequently, the API requirements associated with the corresponding stakeholders for GoDaaS are derived. Then, section 3.6 presents a detailed view on GoDaaS platform architecture in support of our requirements. Having the architecture in place enables the proposed business models. As a proof of concept, three business models are described from developers view at section 3.7. In support of our model, we develop a primary GoDaaS consultation service for developers. The running prototype[5] is detailed at section 3.7.2. Finally, section 3.8 concludes the chapter and presents an outlook on future research directions.

[4]http://open311.org
[5]http://soheil4tuwien.github.io/GoDaaS/tool.html

3.2 Related work

In relation to our work, there are some prominent studies on capturing business models for open government data. The European commission prepared a study on business models for Linked Open Government Data (LOGD) for the ISA[6] programme by PwC EU Services in late 2013 [21]. In this report, the authors provide a theoretical framework to analyze the LOGD. Fourteen entities who offered publishing, linking and accessing open government data as a service were selected as case studies for further analysis. The framework is structured according to the nine areas in the Business Model Canvas (BMC) [16]. There are considerable studies on how to provide government's data as *Linked Data*[7][22]. In [23], [24], [25], [26] and [27], the authors propose a semantic approach on attaching meaning to government data by applying ontologies to formally and semantically represent data.

This chapter outlines a set of abstractions for serving governmental data. Governments produce data that members of the public are entitled to access but format, size, and technology hurdles often prevent such access. GoDaaS is a proposal to mandate that all government data be made available in a form that can be accessed through a unifying set of programming abstractions.

3.3 Government Data Compute Units (DCU)

In the current Gov. 2.0 movement, governments are unable to feed developers with formalized and structured data. Government data is available in data catalogs or APIs, which are not published as a unit, but rather accessible on the data portal. For instance, the *data.gov* and its CKAN[8] API only contain meta-data about datasets. This meta-data includes URLs and descriptions of datasets, which is not handy for programmers. As a reaction to this complexity, we designed a new abstraction layer called *Data Compute Unit (DCU)*, that allows governments to express their data packages more structured and consistent, so that developers can utilize these packages by treating them as objects. DCU copes with an increasing heterogeneity of state government data sets, by providing a unified interface on top of diverse data schemata from various states. In this context, every state government provides its data in a unified interface of DCU. Then external systems, like SOAP services or even programmers are able to invoke an API from DCU library like *transformData().*

We define a DCU abstraction as a GoDaaS platform programming construct, containing raw data and associated meta-data together with its utilization APIs. DCUs can be considered as a programming construct, like Classes, for developing data-intensive applications. Civic developers can compose, program and configure those DCUs to a package like Linked Compute Units[28] and expose them to services delivered in mobile

[6]Interoperability Solutions for European Public Administrations (ISA)

[7]Tim Berners-Lee view: http://www.w3.org/DesignIssues/LinkedData.html

[8]http://ckan.org/

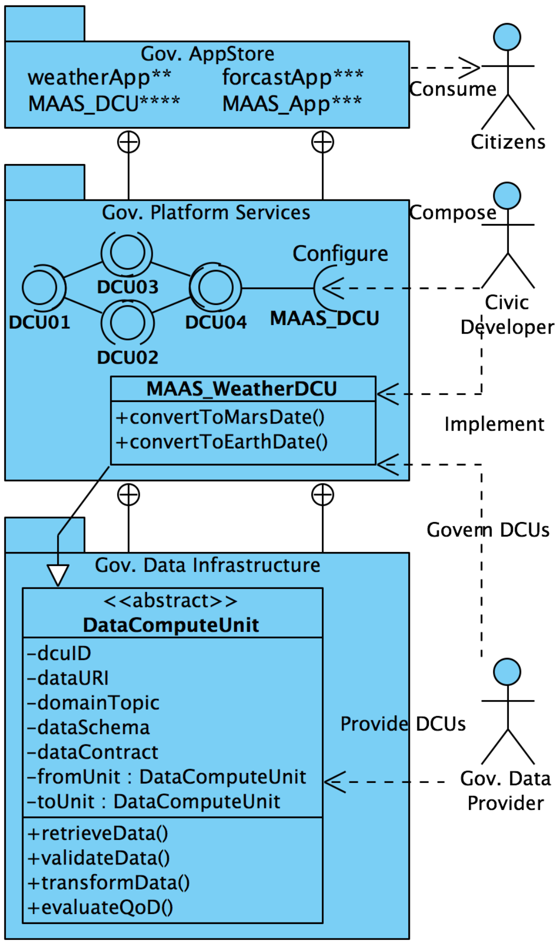


Figure 3.1: GoDaaS Stakeholders together with their relationships.

apps to citizens. As shown in Figure 3.1, DCUs are composed of two parts: The DCU meta data and set of APIs. The meta data provides: (i) a unique ID for future object referencing, (ii) an URI that identifies the data source, which is actually a URL that supports the data schema protocol for the retrieval purposes, (iii) a schema for an structured data extraction and loading, (iv) a contract where terms and usage licenses are detailed, and (v) two *fromUnit* plus *toUnit* elements to wire DCUs together for composition purposes. DCU provides programmatic access to the contents and meta-data of the government data repository and fosters re-use for programmers. We believe governments must provide underlying data resources as *DCUs*.

In this study, our resource unit granularity is a *Gov. Data Compute Unit*. Developers pull and program these units from government data infrastructure and expose DCUs as web services or mobile apps. For instance, NASA has developed an open source REST

API, called MAAS[9], to provide information on the weather data being transmitted by the Curiosity Rover on Mars. We consider the MAAS API together with its meta data as *MAAS_WeatherDCU*. Developers can program and override this unit by building mobile weather apps or analytic applications to utilize the weather data for their research purposes. The MAAS API is available as an open source project under the Apache license[10].

Listing 3.1: Excerpt of DataComputeUnit Class Implementation

```
// Sample MAAS DCU class extending the abstract
// DataComputeUnit class

public class MAAS_WeatherDCU extends DataComputeUnit{
private XML data;
public MAAS_WeatherDCU(){
dcuID = new GUID();
dataURI =
"http://marsweather.ingenology.com/v1/../";
dataSchema = define XSD schema; }
public XML retrieveData(){
data = curl -X GET uri;
return data; }
public bool validateData(){
return dataSchema.validate(data); }
public JSON transformData(){
return JSON.convert(data); }
public float convertToFahrenheit(){
int cels_temp = data.getTemprature();
return (cels_temp * 9 / 5) + 32;} }
```

Listing 3.1 provides a closer touch of programming a DCU by the pseudo-code of some implemented procedures. The *MAAS_WeatherDCU* class implements abstract methods of *DataComputeUnit* class. In the class constructor, all *dcuID*, *dataURI*, and *dataSchema* attributes are initialized. The *dcuID* is of the GUID type, which makes it unique in the whole ecosystem. This unique value is useful for tracing and logging DCU objects. The *dataURI* points to the data source location for data retrieval, which can be an internal service. The *dataSchema* is defined as an XSD file in order to validate the quality of data variable structure. Invoking *retrieveData()*, fetches the data provided in URI address using curl command and stores it as XML format in the *data* variable. The *validateData()* procedure handles schema validation using XSD validation. Next, *transformData()* is a simple function for converting the *data* into JSON format. In the context of *MAASWeatherDCU*, the *convertToFahrenheit()* method fetches the stored temperature data and converts it to Fahrenheit. It implements a light-weight logic regarding to MAAS_WeatherDCU.

[9]http://marsweather.ingenology.com
[10]https://github.com/ingenology/mars_weather_api

The GoDaaS architecture employs an enterprise service bus for its messaging, service integration and orchestration of processes. In Listing 3.2, we show the pseudo-code of core methods of Government Service Bus (GSB). Each *service* subscribes for a specific domain *topic*. The *GovServiceBus* discovers the queried *topic* in its governance repository and creates a new one if it is not already defined. Then, the *service* is added to the instantiated *topic*. The *publish()* function stores the DCU in the service bus *topic* queue. Using the *mediate()* function, the *dcu* objects stored in the *queue* are processed one by one. Based on the topic of the *dcu* object, subscribed *services* are discovered. Having these *services* identified, GSB checks whether the *service* has access to the *dcu* object. Finally, GSB delivers the *dcu* object by calling the *receive()* function of the *service*.

Listing 3.2: Excerpt of Gov. Service Bus Class Implementation

```
// Sample Government Service Bus (GSB) class
// dealing with DCU objects.

public class GovServiceBus {
public void subscribe(Service service ,Topic topic){
topic = find topic in the repository ,
create if not exist
topic.addSubscriber(service); }
public void publish(DataComputeUnit dcu){
queue.push(dcu); }
private void mediate(){
DataComputeUnit dcu = queue.pull();
for all dcu.topic.getSubscribers(){
authenticate(subscriber);
subscriber.receive(dcu);} } }
```

3.4 Stakeholders in GoDaaS

To investigate and elicit the requirements, we classify core stakeholders into three main groups of *Government Data Provider*, *Civic-minded Developers* and *Citizens*. Their dependencies are illustrated in Figure 3.1.

◇ *Government Data Provider*: This entity owns and provides the data. They provision an operating open DCU infrastructure together with the development platform.

◇ *Civic-minded Developers*: The civic developers implement the logic aspects of their *DCU-based* applications. They can program DCUs using the associated APIs for an intended behavior.

◇ *Citizens*: A citizen is the government service consumer. The government body opens up its data to its citizens for more transparency over their services. Citizens eventually may consume the data via mobile applications on the government *AppStore*.

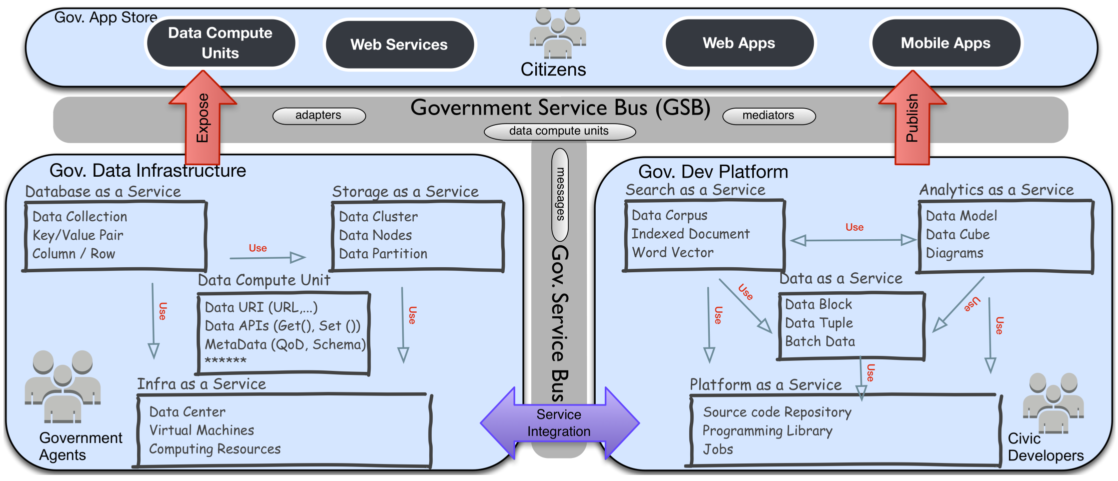


Figure 3.2: GoDaaS system architecture in layers.

3.5 API Requirements for GoDaaS

The *Eight Open Government Data Principles*[11] document outlines the key requirements for open government data. Embracing these eight principles, we delve into our perception of the APIs and derive technical requirements for the Cloud-based government platform (GoDaaS). In our approach, the government is to offer the GoDaaS platform and a set of APIs, so that developers consume those APIs for their application programming.

3.5.1 RQ1. End-to-End Governance Coverage throughout Data Compute Unit Life Cycle:

Governance enforces government policies, governing all aspects (e.g., manipulate, compose, expose, evaluate and control) of the DCUs throughout its life cycle. The stages of DCUs life cycle, through which published-data passes, can be sequenced as collection, processing, use, storage, application, provision and disposition.

3.5.2 RQ2. PaaS-enabled DevOps Integration for Government DCU-based App Programming:

This requirement deals with enabling the development environment, composition, adoption and use of DCU-based apps. As such, it incorporates full development life cycle tooling in support of application programming, debugging, testing, building and deploying processes. Government must provide the open DCU programming interfaces to its data.

[11]https://public.resource.org/8_principles.html

3.5.3 RQ3. Discover, Subscribe and Provision Assets/Apps through a Government AppStore Interface:

Developers compose and program DCUs and expose them as services or mobile apps. Subsequently, the AppStore publishes these apps as marketable entities; then citizens are able to subscribe, keep them in use and pay as they go. This leads to more economies of scale for all parties involved. GoDaaS asset store not just enables the on-demand and automated provision of these apps, but also couples with clients' satisfaction and app recommendation service that meets citizens' requirements.

Stakeholders ↓ Requirements →	RQ1	RQ2	RQ3
Civic-minded Developer		•	•
Government (Federal, State)	•		•
Citizens (Crowd, Individual)			•

Table 3.1: Stakeholders/Requirements relation in GoDaaS.

As the stakeholders and requirements relationship is listed in Table 4.1, the need for asset/app store by each stakeholder stands out. In support of these requirements, we propose the GoDaaS platform. Next, we detail the GoDaaS layered architecture design, indicating the logical separation or division of components.

3.6 GoDaaS Architecture

The impetus behind GoDaaS platform is how to adopt a data-driven cloud-enabled platform, which provides interfaces for developers and private sectors to develop their publicly available applications that expose the underlying data.

Figure 3.2 illustrates the GoDaaS architecture, which encompasses in four major layers: (i) *Gov. Data Infrastructure*, (ii) *Gov. Development Platform*, (iii) *Gov. Service Bus* and (iv) *Gov. AppStore*. Using Gov. data infrastructure, the government incorporates cloud computing technologies to virtualize its data resources into *DCUs*. This enables elastic DCU provisioning, meaning that government is able to dynamically allocate DCUs to developers in an on-demand fashion. Conversely, the apps can pull DCUs by invoking storage or database services to complete a transaction. Along with this layer, the Gov. development platform layer is in place, where authorized developers can implement and deliver their applications on top of government data. This platform enables programming, composing and deployment of DCU-based applications. It is basically a government-class cloud platform, which employs data-intensive programming models supported by integrated development libraries and services like data analytics and search facilities, which will empower the development process. On top of these layers, the government AppStore stands out. The AppStore supports multi-tenancy, where identities like state governments, third parties and civic developers are authorized to deploy their applications for trading. Through governance services, the AppStore ensures that deployed apps comply with the government policies and regulations respectively. Moving forward, the monitoring, metering and billing services track the asset/app usage and debit citizens.

GoDaaS multi-layer architecture needs an integration enabler for its implementation and a uniform service delivery model. Government service bus (GSB) layer glues all the entities, agencies and services together through its messaging and queuing mechanisms. Together with the integration, GSB also provides data adapters to the DCUs registry, which is a point of access to all provided and customized DCUs by governments or developers. It acts as an intermediary component to accept and provision the requests to DCUs. Then it invokes the associated adapter to retrieve the DCU, validates and transforms it through data schema and forwards it to the developer, for instance.

In order to catalyze the adoption of cloud delivery models, we have come up with seven business models for the government open data. GoDaaS can include the following types of data-driven business models: *Data Infrastructure as a Service (IaaS)*, *Storage as a Service (STaaS)*, *Data as a Service (DaaS)*, *Database as a Service (DBaaS)*, *Platform as a Service (PaaS)*, *Search as a Service (SEaaS)* and *Data Analytics as a Service (DAaaS)*. All seven business models and their associated elements are consistently specified and modeled in details, mostly from the developer's view.

In this context, business modeling is driven by the value of API calls chain, as a sequence of service invocations to exchange DCUs among governments and developers. This value can be realized through developed mobile applications, that are consumed by citizens. For modeling purposes, we consider two main languages: the Business Model Ontology (BMO)[12] and the OMG Business Motivation Model (BMM)[13].

3.7 GoDaaS Business Models

Due to space limitations, we only present *Database as a Service* and *Platform as a Service* business models in the chapter. Since DCU is an abstraction layer between data corpus and developers, we integrate the DCU with several business models to help civic-minded programmers make better use of GoDaaS framework. All the interactions among business models in GoDaaS are based on the unified interface exposed by the DCU. We briefly recap the three models here, and refer the reader to the GoDaaS site[14] for further details of all seven implemented business models.

3.7.1 Gov. Database as a Service (DBaaS)

Government's parallel computation and distributed storage requirements to perform data-intensive analytic processes on their big data, motivate the use of elastic and scalable

[12]http://www.businessmodelgeneration.com/canvas
[13]http://www.omg.org/spec/BMM
[14]http://soheil4tuwien.github.io/GoDaaS/index.html

NoSQL databases such as Apache Hbase[15] or Casandra[16].

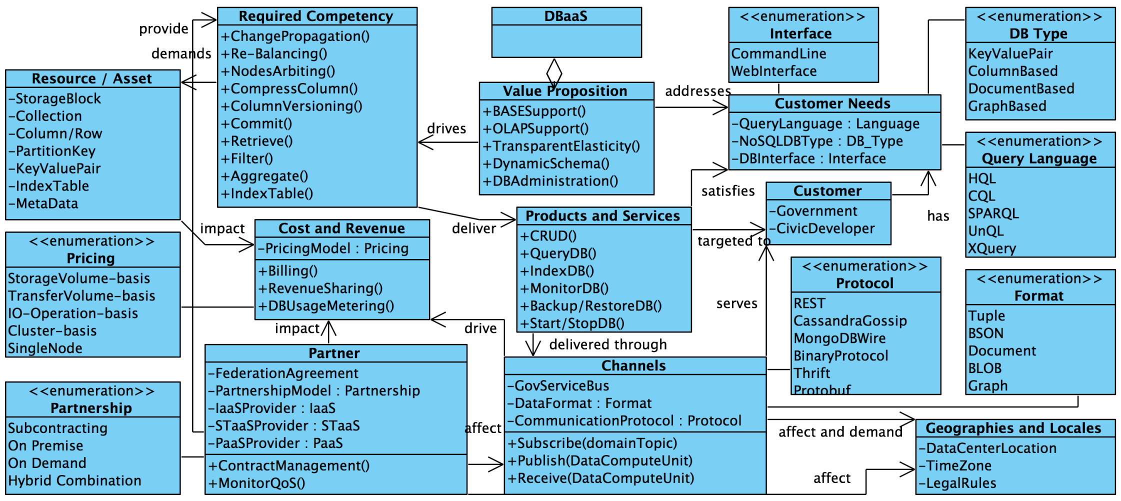


Figure 3.3: GoDaaS Database as a Service (DBaaS) business model class diagram.

DBaaS as illustrated in Figure 3.3 allows federal governments to provide a single logical database to span geographically dispersed state governments' data centers, while maintaining intelligent load balancing. Storage load is provisioned on IaaS clouds in the form of virtual disks that can be attached and detached from running VM instances[29]. Dealing with such state-aware loads means enabling storage nodes to scale elastically, tightly interfaced with the STaaS and IaaS layers. Eventually the GoDaaS platform will expose database services as APIs to developers and will govern patterns of API usage.

3.7.2 Gov. Platform as a Service

Government PaaS implements an application factory platform, enabling developers to *Code-on-Demand*[30]. The platform then incorporates the state or federal SLAs to ensure the government applications which are built through this process, follow a set of enforced guidelines. Data intensive government application development often involves large volumes and distributed data sets. Hence, the platform provides data-intensive parallel programming models to provide distributed control of code execution during job enactment. For instance, in the *MapReduce* programming model, data processing is broken up into distributed fine-grained *Map* and *Reduce* tasks to devise a parallel solution, performed by a *Master-Slave* design pattern.

As shown in Figure 3.4 the government PaaS objective outlines the need for a *"Collaborative & Composable Platform"*, where the federal government would need to leverage "sharing" of DCU resources and fork open source applications among state governments for civic developers. This contributes to sharing best development practices and lessons

[15]HBase Homepage. http://hbase.apache.org
[16]Casandra Homepage. http://cassandra.apache.org

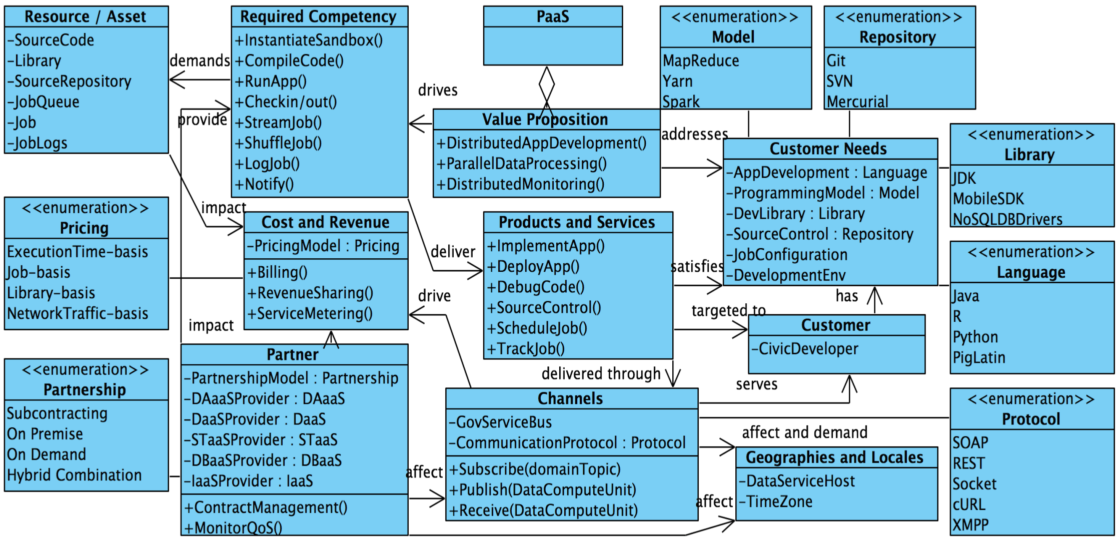


Figure 3.4: GoDaaS Platform as a Service (PaaS) business model class diagram.

learned from current celebrated *"government-ready"* mobile applications. Sharing such code artifacts leads to agility in the development cycle. The platform provides a collaborative development environment (using SVN, Git, etc.), accompanied with a set of programming APIs like *compileCode()*, *debugCode()*, etc., and some development libraries for seamless coding. The developers can import the available DCUs from the registry into their IDE, then program and compose configuration and code artifacts into a single development project and deployment archive called composite application.

3.7.3 Data Analytics as a Service

An in-depth analysis of government data will boost its services delivery to the respected citizens. Government data agents and services amass underlying data and publish analytic APIs for developers to make more sense of data. By analyzing the exposed data and wrapping the analytic processes into application understand and analyze their operational and transactional data. Civic developers program DCUs using such rich analytic APIs to derive values from the data to glean insight into the behavior. Federal and state governments benefit from such applications to make decisions based on discovered patterns and anomalies.

From the Fig. 3.5, the data analytic services utilizes DaaS partnership to retrieve and aggregate data through exposed DCUs. The aggregated data then stored to a distributed file system like Hadoop[17] cluster using DBaaS and STaaS service providers. To deliver analytics on this distributed data, DAaaS applies correlative and data mining algorithms like K-Means through a machine learning library like Mahout[18] to discover new knowledge

[17]Hadoop Homepage: http://hadoop.apache.org
[18]Mahout Homepage: https://mahout.apache.org

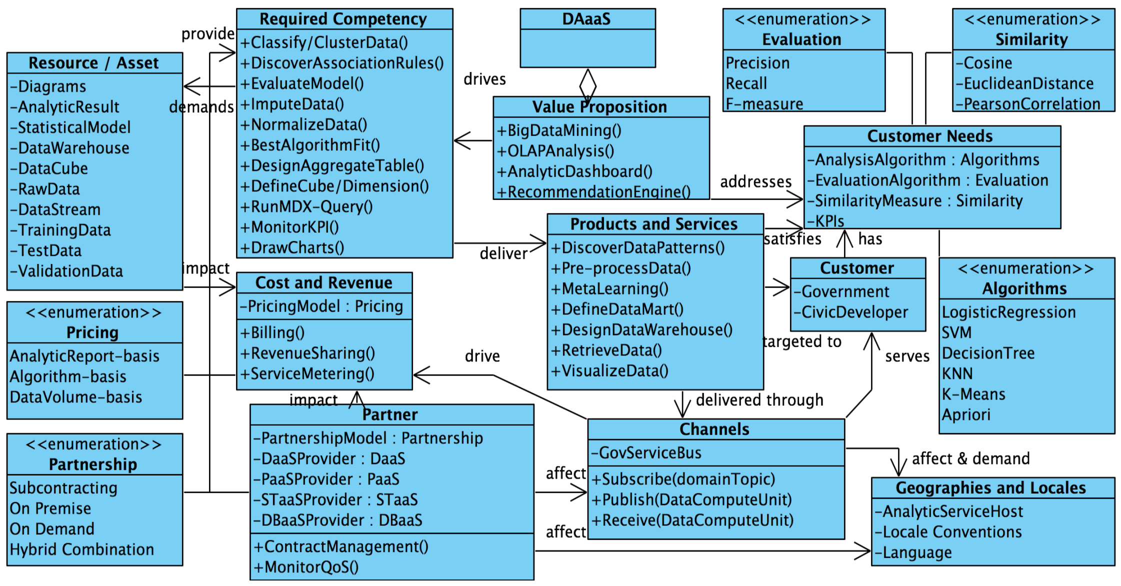


Figure 3.5: GoDaaS Data Analytics as a Service (PaaS) business model class diagram.

on-demand. Furthermore, this enables an analysis tasks in the applications to be executed at a computational resource close to where the required data set is.

One interface for the citizens to consume and take action based on the analysis results and KPI alerts is visualization. DAaaS visualization APIs supported by charting libraries will visualize the exposed objects on the dashboard. Hence, the objective of visualization services is to publish the summary of the analytics on to dashboards.

As a prototype, we have developed a consultation service where developers can clarify their application specification and receive advice on their application's technical requirements for it's commercial possibilities. The aim of the tool is to design a tailored business model conforming to the needs of a new application. As a proof of concept, the service forms a business model for the DAaaS. The service is fed with a sequence diagram 3.6, also available in the referenced project repository, demonstrating the internal relations between the methods and attributes of the DAaaS business model. The service parses the sequence diagram and traces model element dependencies for new model generation. The tool is hosted and running on the GoDaaS site.[19]

3.8 Conclusion and Outlook

This chapter presents a abstraction that combines government data URL and its associated interface called DCU to enable open government data as a service. The DCU also incorporates the government policy. Based on DCU, the authors expand the discussion to the opportunity of civil use of the government data and its potential advantages.

[19]http://soheil4tuwien.github.io/GoDaaS/index.html

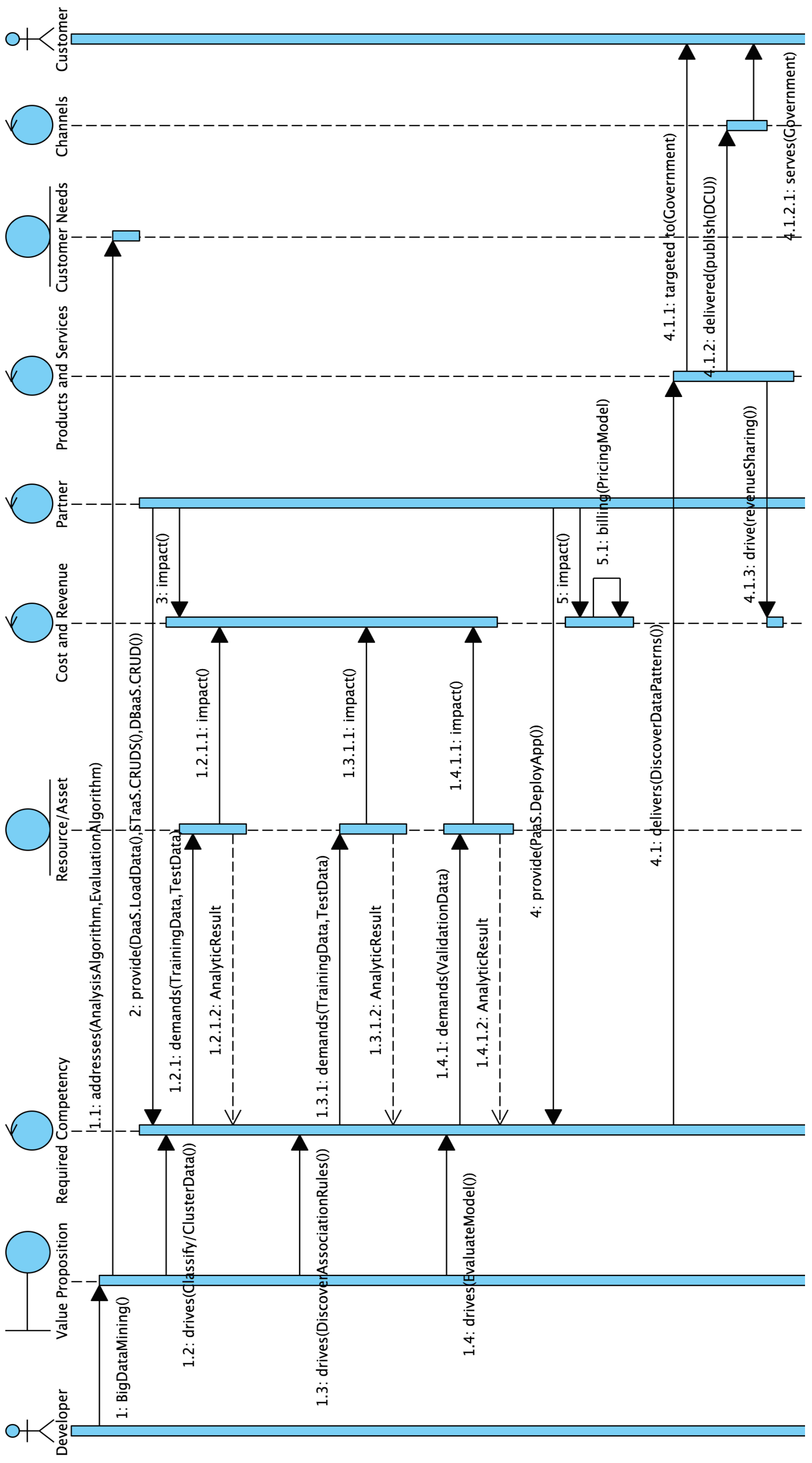

Developer
Value Proposition
Required Competency
Resource/Asset
Cost and Revenue
Partner
Products and Services
Customer Needs
Channels
Customer
1: BigDataMining()
1.1: addresses(AnalysisAlgorithm,EvaluationAlgorithm)
2: provide(DaaS.LoadData(),STaaS.CRUDS(),DBaaS.CRUD())
3: impact()
1.2: drives(Classify/ClusterData())
1.2.1: demands(TrainingData,TestData)
1.2.1.1: impact()
1.2.1.2: AnalyticResult
1.3: drives(DiscoverAssociationRules())
1.3.1: demands(TrainingData,TestData)
1.3.1.1: impact()
1.3.1.2: AnalyticResult
1.4: drives(EvaluateModel())
1.4.1: demands(ValidationData)
1.4.1.1: impact()
1.4.1.2: AnalyticResult
4: provide(PaaS.DeployApp())
5: impact()
5.1: billing(PricingModel)
4.1: delivers(DiscoverDataPatterns())
4.1.1: targeted to(Government)
4.1.2: delivered(publish(DCU))
4.1.3: drive(revenueSharing())
4.1.2.1: serves(Government)

Figure 3.6: GoDaaS Data Analytics as a Service (DAaaS) sequence diagram.

In support of such abstraction layer, we present a reference architecture to enable the development of GoDaaS, that is, Government Data as a Service. The goal of the architecture is to provide a formal way for a given government to expose open-data, in a way developers with civic-minded ideas can easily reuse it instead of spreadsheets or PDF files as classically used. The key idea is to expose mechanisms that will allow developers to public application in an app store, on top of governmental datasets.

So far, we modeled seven possible business opportunities on government open data. In order to take such opportunities, governments should seek to ease any friction that limits developers' ability to build their tailor-made applications on top of these business models. We have proposed that governments provide their underlying data resources as Data Compute Units (DCUs) for the sake of seamless development. Civic developers compose and program those DCUs and expose them to services delivered in mobile Apps to citizens. We envision Gov. Data Compute Unit to gain acceptance as a *de facto* standard of an open government data resource unit norm, through broad adoption. In this context, we extended our contribution to an open government data service model entitled *GoDaaS* as a government data processing platform, where civic programmers are motivated to develop applications for citizens. As an outlook, we plan to elaborate and extend the GoDaaS to build a development blueprints tailored for a specific application that logically combine, orchestrate, and consume government data services.

CHAPTER 4

Cloud Manufacturing

4.1 Introduction

Industrial sectors now seek for distributed manufacturing control systems that provide satisfactory, adaptable and robust systems, rather than optimal solutions that require several hard assumptions to be met [31]. They are inspired by the convergence of utility-driven opportunities of novel distributed technologies, such as cloud computing and Internet of Things (IoT), coupled with collaborative engineering and applications that are transforming the manufacturing processes like supplement, consumption, and production models over the web. This contributes to the ability of dynamic provisioning of elastically scalable manufacturing resources. Cloud technologies efficiently enable integrating geographically diverse production lines to share knowledge and collaborate through virtualization layers and well-defined interfaces. Thereby, this model facilitates the inter- and intra-factory communication and collaboration in cloud manufacturing environment. In this context, the distributed execution of shop floor jobs is applicable. Underneath all the technologies involved, the essence of cloud computing is the cornerstone of distributed and sustainable manufacturing. According to the definition of National Institute of Standards and Technology (NIST) [3], cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. As thus, a cloud Manufacturing system is capable of serving multiple companies to planning, control and collaboration over the web. It makes production planning and manufacturing work in progress (WIP) distributable, controllable, composable and portable. In comparison to traditional production services, cloud manufacturing is more complex and encompasses a broader diversity of services to adapt to ever-changing *customer-specific* requirements. In particular, CloudMan services are subject to realize concepts of “Manufacture on-Demand“ and “Manufacturing as a

Service“ to meet unforeseeable events such as change of orders in an on-demand fashion.

Moving forward, the aim of this chapter is firstly to conduct a detailed analysis on the CloudMan platform stakeholders; then to identify and present a set of architectural API requirements to meet the stakeholders needs in such environments. Secondly to propose an adequate CloudMan architecture by presenting the layers, including its service abstractions, with their associated core services. Lastly, we describe the major components and how they interact with one another to fulfill each requirement. CloudMan handles dynamic configuration, portability and deployment of manufacturing services across multiple collaborating manufacturers, by incorporating TOSCA standard. Henceforth, the product Bill of Material (BOM) is virtualized on the cloud and is referred to as Bill of Manufacturing Services (BOMS). The product designers model their products TOSCA-based BOMS by composing both physical manufacturing resource, e.g. devices, machines, sensors, materials, and the unphysical services, computing resources and capabilities e.g. product documents, data and required software services, device drivers using TOSCA standard. CloudMan deploys the distributed manufacturing processes and plans of such TOSCA-based BOMS. The chapter ends with certain conclusions, focused on explaining the applicability of the proposed architecture for cloud manufacturing systems. However, at the moment, to the best of our knowledge, its not trivial for cloud and business developers to fully utilize manufacturing resources from multiple factories to realize the cloud manufacturing platform.

To this end, our contribution is twofold: (i) Incorporating OASIS TOSCA standard for modeling cloud manufacturing Bill of Materials (BOM). (ii) A portable cloud manufacturing platform architecture together with its big data processing model.

The chapter continues with the specification of portable cloud manufacturing services at section 4.2. Next, we model the TOSCA-based Bill of Manufacturing Services (BOMS) at section 4.3, to demonstrate the feasibility and applicability of TOSCA in our proposed framework. Section 4.4 is devoted to the core stakeholders and their relationships in cloud manufacturing ecosystem. At section 4.5, the API requirements associated with the corresponding stakeholders for CloudMan are derived. Subsequently, section 4.6 presents a detailed view on CloudMan platform architecture, and a focus is put on implementing this new model with the API requirements and the supporting system architecture. Then, section 4.7 presents the platform big data processing architecture. Section 4.8 surveys some scientific related work. Finally, section 4.9 concludes the chapter and presents an outlook on future research directions.

4.2 Portable Cloud Manufacturing Services

We define cloud manufacturing as a ‘*distributed manufacturing execution model, where underlying resources envisaged in the Internet of Things (IoT), are elastically exposed and utilized as cloud services, then composed and orchestrated for a manufacturing task in an on-demand fashion*’. The cloud virtualization technology abstracts away the complexity of underlying manufacturing physical resources and their associated operations from the product designer and developer. The IoT, takes the responsibility of controlling the sensor network and its data and event flow with the closer touch on shop floor operations. Collaborative engineering, addresses the orchestration and collaboration side of the manufacturing flows. Then the manufacturing services and encapsulated resources (devices, sensors, materials, drivers, etc.) are composed into the product BOMS that can be accessed, configured, invoked, deployed and coordinated on distributed production lines in a near real-time manner.

Listing 4.1: Excerpt of a TOSCA-based Cloud Manufacturing Plan

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Plans>
<Plan id="DeployManufacturingService"
name="Car_Manufacturing_Build_Plan(CMBP)"
planType="http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">

<Precondition expressionLanguage="www.example.com/text"> ?
Invoke if Order is Paid.
</Precondition>

<PlanModel>
<process name="Produce_10_BMW_X6" id="p1">
<documentation>
Deploy new instance of the CMBP to produce 10 BMW cars.
</documentation>

<task id="Task1" name="Check_the_Inventory_for_BOMS_Resource_availability"/>

<task id="Task2" name="Ship_materials_to_shop_floor_from_warehouse"
isSequential="true"
loopDataInput="Task2Input.CarCounter"/>
<documentation>
Assumption: Task2 gets 10 as an Input, sets the CarCounter that
indicates execution number of task.
</documentation>

<task id="Task3" name="Assemble_Car_Items"
isSequential="true"
loopDataInput="Task3Input.CarCounter"/>

<task id="Task4" name="Colour_the_Car"
isSequential="false"
loopDataInput="Task4Input.CarCounter"/>

<sequenceFlow id="s1" targetRef="Task2" sourceRef="Task1"/>
<sequenceFlow id="s2" targetRef="Task3" sourceRef="Task2"/>
<sequenceFlow id="s3" targetRef="Task4" sourceRef="Task3"/>
</process>
</PlanModel>
</Plan>

<Plan id="Halt_Car_Production"
planType="http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
planLanguage="http://docs.oasis-open.org/wsbpel/2.0/process/executable">
<PlanModelReference reference="Mfg:HaltMfgSrv"/>
</Plan>
</Plans>
```

The monitoring service instances are responsible for sensing and interpreting the operational data received from the deployed manufacturing services across various units. If malfunctioning detected, the monitoring instance resumes the corresponding services, changes the context status and invokes recovery plan. One potential fault-recovery process can be incorporating portable cloud manufacturing services. Being able to port the whole package of dependent manufacturing services, serves better proactive maintenance of defects detected in one production line. The fault-recovery process will port the BOMS to other corresponding and authorized production lines and resume the job completion.

4.3 TOSCA-based Bill of Manufacturing Services

The TOSCA-based BOMS of a configurable product contains all the components that are required to manufacture the product. It is composed of comprehensive list of raw materials, components and assemblies required to build or manufacture a product. BOMS is usually in a hierarchical format, with the topmost level showing the end product, and the bottom level displaying individual components, resources and materials. Each item in BOMS describes the relationship between a parent (assembly) item and a child (component) item. In order to establish a cross-factory manufacturing governance and ultimately integrate distributed production lines, a standard shall be used to define product structure covering the relationship between objects, components and their configuration recipes.

There are couple of standards, which each does utilize the data exchange but lacks supporting the cloud and portability features. PDX[1] is the *Product Data eXchange* standard for the e-supply chain. Its standardization effort is focused on the problem of communicating product content information between Original Equipment Manufacturers (OEMs), Manufacturing Services providers and component suppliers. The standard is based on XML as a flexible way to encode structured data into a format that is both human and machine-readable. It provides a way to describe product content (BOM, Approved Manufacturer Lists (AML), Drawings, etc.), Engineering Change Requests (ECR), Engineering Change Orders (ECO) and Deviations in an XML format. This enables a total product definition to be described at a level appropriate to facilitate supply chain interactions. The standard is designed to transfer technical information including BOMS, AML, as-built product configuration, and change (Engineering, Manufacturing, Product) information, as well as manufacturing features, tolerance specifications, material properties and finish specifications. Another more in-use standard, called the Business-to-Manufacturing Mark-up Language (B2MML)[2] is an XML-based standard maintained by the World Batch Forum (WBF), which specifies data formats for information exchange between different enterprise control systems [B2MML03]. It provides an implementation of the ISA-95 standard in terms of XSD schema definitions.

Looking forward, having the essence of cloud computing in manufacturing domain, leads to an evolution in its manufacturing service delivery models. Cloud-enabled manufacturing services can be identified, specified, designed, composed, configured and realized or manufactured on-demand. Noted this, the BOMS can now be considered as cloud manufacturing virtual product, applicable to be deployed in multi-production lines,

[1]http://webstds.ipc.org/standards.htm#x2570

[2]B2MML (2003): Business to manufacturing markup language. The World Batch Forum, http://www.wbf.org

resulting in actual “as-built” assets. In this respect, the need for a cloud standard or specification to represent BOMS seems to be vital. The following offerings are observed and the appropriate one is ensured.

Cloud standards like TOSCA and OVF are for packaging the cloud services and resources as marketable entities. OASIS TOSCA approaches composing the cloud resources from the design perspective, rather than the actual resources that support the application, as OVF does. The OVF structure, on the contrary, emphasizes on what must be installed and how. TOSCA refers to OVF packages as deployment artifacts. A component of a TOSCA service can be implemented by deploying an OVF package. TOSCA’s approach is higher level than OVF. In fact TOSCA represents a business-oriented development methodology, which can be a significant achievement in IT-business alignment. Thus, TOSCA is a better choice in our mappings.

The TOSCA language introduces a grammar for describing service templates, by means of *Topology Templates* and *Plans* in which can be utilized to define the BOMS as well. The focus is on design time aspects, i.e., the description of manufacturing services, resources, materials or in broader view, “things” to ensure their exchange. The runtime aspects of TOSCA are addressed by providing a container for specifying models of plans. In cloud manufacturing case, these plans can address supply chain plans, production plans, maintenance plans, etc. which can support the management of instances of manufacturing services. In fact, the designers will be more focused on the TOSCA-based BOMS topology design, which is a logical relationship between product components, equipments and assembly items. The TOSCA-based BOMS topology defines its item’s specification regardless of any explicit development and manufacturing operation details. With this approach, the manufacturing operations of a product can change without any effect to its BOMS design. Hence, both the manufacturer and the user are able to compare various actual manufacturing solutions and in terms of cost, quality and timely production to make a reasonable choice. Figure 4.1, shows the meta model of the TOSCA-based BOMS. The root of a TOSCA-based BOMS is the *Product Template.* The product template contains a directed graph that represents the structure of the product called a *Product BOMS Topology.* Every product template has at least one product TOSCA-based BOMS topology. The topology graph is composed of nodes and edges. Edges in a directed graph are links with a direction from node to node. The edges in a product BOMS topology graph are binary relationships between nodes. The nodes represent the logical components, items or objects of the product. These nodes and relationships are templates that are patterns for the actual resources, objects or materials and their relationships instantiated in a deployed manufacturing service. Relations capture, represent and quantify associations between objects. Plans orchestrate various aspects of a manufacturing service life cycle. The TOSCA specification defines *Build Plans* and *Termination Plans.* Build plans orchestrate the deployment, installation and production operations of a BOMS. Listing 4.1 implements an excerpt of a TOSCA-based BOMS build plan in cloud manufacturing. The complete implementation of a TOSCA-based Bill of Manufacturing Services (BOMS) is available on the CloudMan

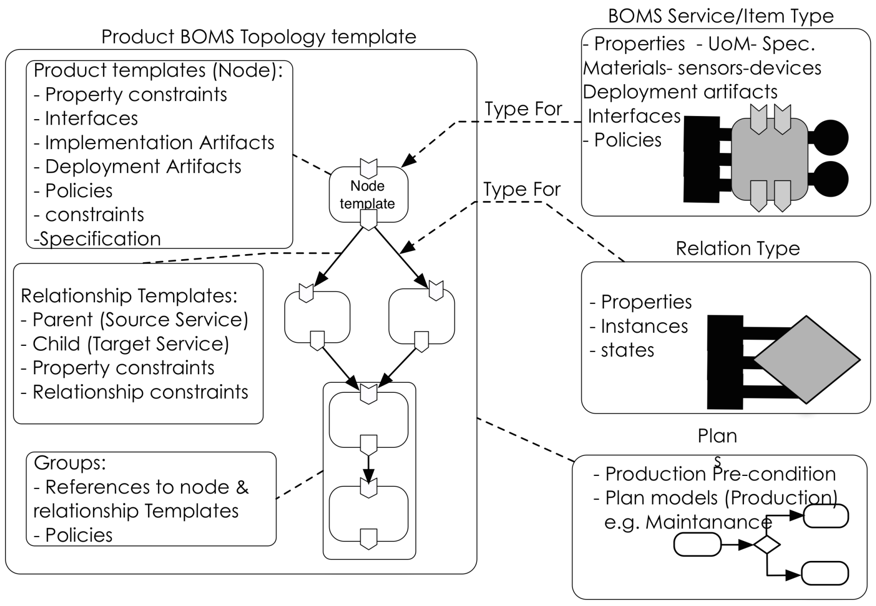


Figure 4.1: The core elements of TOSCA-based Bill of Manufacturing Services (BOMS) Template and the Topology.

site[3].Termination plans orchestrate decommissioning a production line. Designers of TOSCA-based BOMS can add plan types as needed. Therefore TOSCA-based BOMS encapsulates the required information for product life-cycle management. The utility of TOSCA-based product BOMS maximizes the portability of manufacturing document, and guarantees a smooth data flow.

In this chapter, we propose CloudMan platform which deals with the realization and provisioning of all types of manufacturing resources as services, for all phases of the production life-cycle, from product TOSCA-based BOMS topology design to its distributed deployment. These manufacturing resources such as physical resources, software services and data units are classified in [32] and can be dynamically configured and composed into the TOSCA-based BOMS to be utilized through manufacturing services.

4.4 Stakeholders in CloudMan

To investigate and elicit the requirements, we first detail a comprehensive study about the cloud manufacturing ecosystem stakeholders, who have business interest in this environment. Specifically, their participation roles, the services they provide and the APIs they consume. Their dependencies are drawn in Figure 4.2. However, we need to have a systematic classification of stakeholders in order to understand interfaces that they could be directly consuming. In our work, we classify stakeholders into six main groups of

[3]https://github.com/soheil4TUWien/CloudMan

Original Equipment Manufacturers, *Platform provider*, *Product Developer*, *Manufacturers*, *Coordinators* and *Auditors* as illustrated in Figure 4.2:

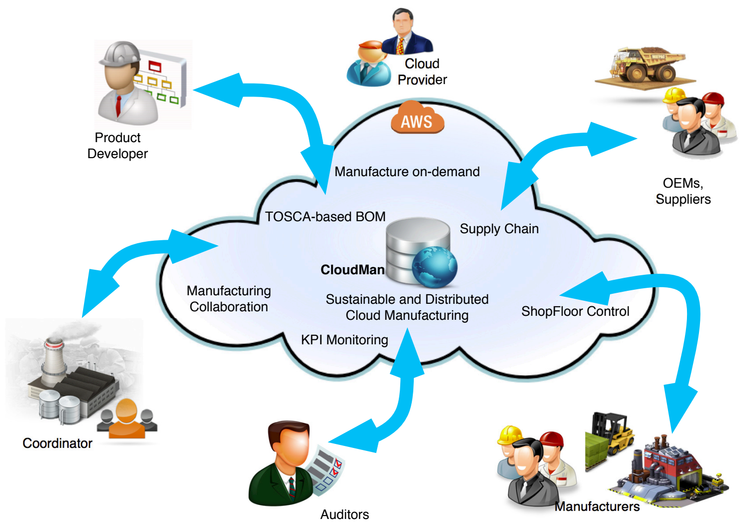


Figure 4.2: Main stakeholders in CloudMan Manufacturing System.

◇ *OEMs*: Original Equipment Manufacturers produce devices or components that can be consumed and utilized within other companies' manufacturing process. Their sustainable equipments can leverage and optimize the distributed manufacturing process. They can also provide logistics services and take charge of all the shop floor issues involved.

◇ *Cloud Platform providers*: They offer computing resources for enabling cloud manufacturing solution. They can be categorized as service providers (i.e., WSO2 ESB) or infrastructure providers (i.e., AWS EC2).

◇ *Product Developers*: In order to initiate a manufacturing process, BOMS should be modeled and designed. The product developers or designers process the user requests and then look for the most appropriate and applicable manufacturing resources that fit the request criteria. Then they model their product BOMS by composing both physical manufacturing resource, e.g. devices, machines, sensors, materials, and the unphysical services, computing resources and skills like product documents, product specification and required software services. In our platform, the product BOMS should be modeled using Oasis TOSCA standard and then the TOSCA-based product BOMS will be deployed to the distributed production lines for the actual manufacturing operations. The TOSCA-based BOMS components, will be mapped and deployed to actual services and manufacturing resources, devices and equipments at the production layer which then can be mapped, allocated and deployed on multiple production platforms.

◇ *Manufacturers*: They provide production lines with the associated and required human interactions in manufacturing at the shop floor. This workforce collaboration (required manpower) includes functions such as assembly, testing, quality control. Manufacturers may collaborate as well as compete.

◇ *Coordinators*: They take care of orchestration of cloud manufacturers in terms of serial and simultaneous production lines. In particular, they manage the portability and interoperability of manufacturing resources among multiple manufacturers. They also manage the resource and product delivery by dealing with SLA negotiation between manufacturers and a consumer. Specifically, they are hiding the diversity and heterogeneity of cloud manufacturers to consumers and coordinate a federated manufacturing.

◇ *Auditors*: In principle, they might be a third party partners, who watch and audit the manufacturing performance and the usage of resources on production lines through standard Key Performance Indicators (KPIs). Auditors make sure of SLA and the quality assurance of product delivery.

4.5 API Requirements for CloudMan

Actors↓ Requirements →	RQ1	RQ2	RQ3	RQ4	RQ5	RQ6
Cloud provider (e.g. IaaS, PaaS)	●					●
OEMs (e.g. Devices, Equipments)			●	●		
Manufacturers (e.g. Workforce)		●	●		●	●
Coordinators (e.g. Control flow)	●			●	●	●
Auditors (Observe Mfg flow)				●	●	
Product Designer (Compose BOMS)	●	●	●	●		●

Table 4.1: Relationships between Requirements and Stakeholders in CloudMan.

Our approach is to offer a cloud manufacturing platform and a set of APIs so that the product developers/designers can design their products seamlessly and circumvent the abstraction layers over the production lines and gain access to the original APIs for resource utilization. From the API engineering perspective, we delve into our perception of the APIs for the cloud manufacturing paradigm. To clarify the expectations of the stakeholders as consumers of these APIs, we classify them into three main categories: (a) TOSCA-based BOMS Formation APIs; (b) Resource Match-making APIs; (c) Manufacturing Provisioning APIs. Thus back to the developer of a cloud manufacturing product, they compose and connect diverse services, resources and raw materials from various manufacturers. They also make sure of clients' requirements to be satisfied by the chosen ingredients. we observe the following main API requirements for architecting the CloudMan:

4.5.1 RQ1. APIs Runnable on Multiple Cloud-enabled Manufacturing Systems

The goal is to provide a layer of abstraction between the cloud-enabled product BOMS designers and the multiple production lines, which basically abstracts the differences among diverse manufacturers. Designers compose abstract TOSCA-based BOMS at this layer, which then can be distributed and deployed on multiple production platforms. This will grant the designers to an "*on-Demand*" and "*at-Scale*" manufacturing service provisioning on multiple and distributed production lines with a single platform.

4.5.2 RQ2. Discovering Product Resources and Materials through Query Interface

This requirement deals with manufacturing resource discovery procedure. First, it manages the resource discovery process among all the available manufacturers. Considering the *Publish-Subscribe* design pattern, the manufacturers publish their profiles in a knowledge base or manufacturing registry. Then the discovery agent broadcasts demand for the actual resource availability among multiple manufacturers. This is required for designing TOSCA-based BOMS.

4.5.3 RQ3. Match-making APIs to Map/Utilize TOSCA-based BOMS to Services to Production Resources

This requirement deals with production resource rating and reservation procedure. After having the manufacturing resources discovered, first the agent rates them based on the matching policy and deployment strategy, then chooses the more appropriate vendor/-manufacturer and the resources, wherein the specification fits in upon requirements in terms of quality and cost constraints. Finally, the confirmed BOMS will be deployed for the actual production and will reserve the resource for future manufacturing provisioning on the shop floor. In summary, production resource matchmaking APIs provide users with a list of recommended resources indicating the best match between BOMS requirements and manufacturers' offer.

4.5.4 RQ4. Ensuring end-to-end BOMS-centric Manufacturing Resource Coverage

The impetus behind the sustainable and distributed cloud manufacturing is driven by the dynamics of the market and ever-changing requirements and the need of product developers to optimize costs with the improved quality. Therefore, it is vital to ensure that the recommended and chosen resources and materials follow the constrains based on agreements and supports all the requested BOMS specification with the cost and quality demanded, while avoiding the dependency or lock-in to one vendor/manufacturer.

4.5.5 RQ5. Manufacturing Monitoring through multi-Production Lines Watch Interface

The platform should provide monitoring services for manufacturing flows running on multiple production lines. Developers can benefit of these APIs in order to collect data and track metrics, gain insight, and react immediately to keep their product development running smoothly. The match-Maker agent can also monitor and process custom Key Performance Indicators (KPIs) to gain cloud-wide manufacturing visibility into resource utilization, application performance, and operational health for future production planning.

4.5.6 RQ6. Managed Interoperability of Production Flow among Multiple Factories

The variety of the service interfaces design makes the product development and the hosted production lines dependent together, which poses the interoperability challenges between multiple manufacturing services. We need models for interoperability and orchestration of the services designed for cloud manufacturing environments.

As listed in Table 4.1, CloudMan requirements are directly associated with a stakeholder, who consumes an API and performs a specific operation to the fulfillment of manufacturing requirement.

4.6 CloudMan Platform Architecture

This section presents the CloudMan architecture that realizes the aforementioned core concepts. Before documenting the architecture, we study the term Manufacturing Execution System (MES) defined by the Manufacturing Execution System Association (MESA)[4] as follows: "Manufacturing Execution Systems (MES) deliver information that enables the optimization of production activities from order launch to finished goods. Using current and accurate data, MES guides, initiates, responds to, and reports on plant activities as they occur. The resulting rapid response to changing conditions, coupled with a focus on reducing non value-added activities, drives effective plant operations and processes. MES improves the return on operational assets as well as on-time delivery, inventory turns, gross margin, and cash flow performance. MES provides mission-critical information about production activities across the enterprise and supply chain via bidirectional communications."

As a blueprint for designing the architecture, we briefly review the information flow between the MES and various heterogeneous connected systems as defined by ISA-95[5] standard. The Product Lifecycle Management (PLM) sends product information like

[4]www.mesa.org

[5]ISA-95 is an international standard for developing an automated interface between enterprise and control systems, specifically for global manufacturers.

its bill of materials, work instructions, equipment configurations, operations list and their execution order to the MES for the operational purposes. This information has been defined using B2MML(Business-to-Manufacturing Markup Language)[6]. In our architecture, we model this PLM information using TOSCA standard. The MES sends the manufacturing test results to PLM, then sends the performance results together with produced and consumed resources to the legacy applications like ERP system. On a shop floor side, the work instructions and recipes are send from MES to Programmable Logic Controllers (PLC) or Gateways for actual operations. As a result, production data are sent back to MES. The MES layer, which is responsible for managing the factory, sits below the ERP, manages the business. The industry standard for plant floor connectivity and process control is called Open Platform Communications (OPC). The standard specifies the communication of real-time plant data between control devices from different manufacturers.

The architectural model proposed for the CloudMan platform as illustrated in Figure 4.3, is organized on five interconnected layers: (i) Manufacturing Vertical Applications Layer, (ii) Manufacturing Core Services Layer, (iii) Manufacturing Execution System Layer, (iv) Manufacturing Service Bus Layer and (v) Manufacturing Infrastructure Resource Layer. Next is focused on each layer's specification and capabilities. In the architecture big picture, we also provide a sample WSO2[7] service which can implement the layer's capabilities.

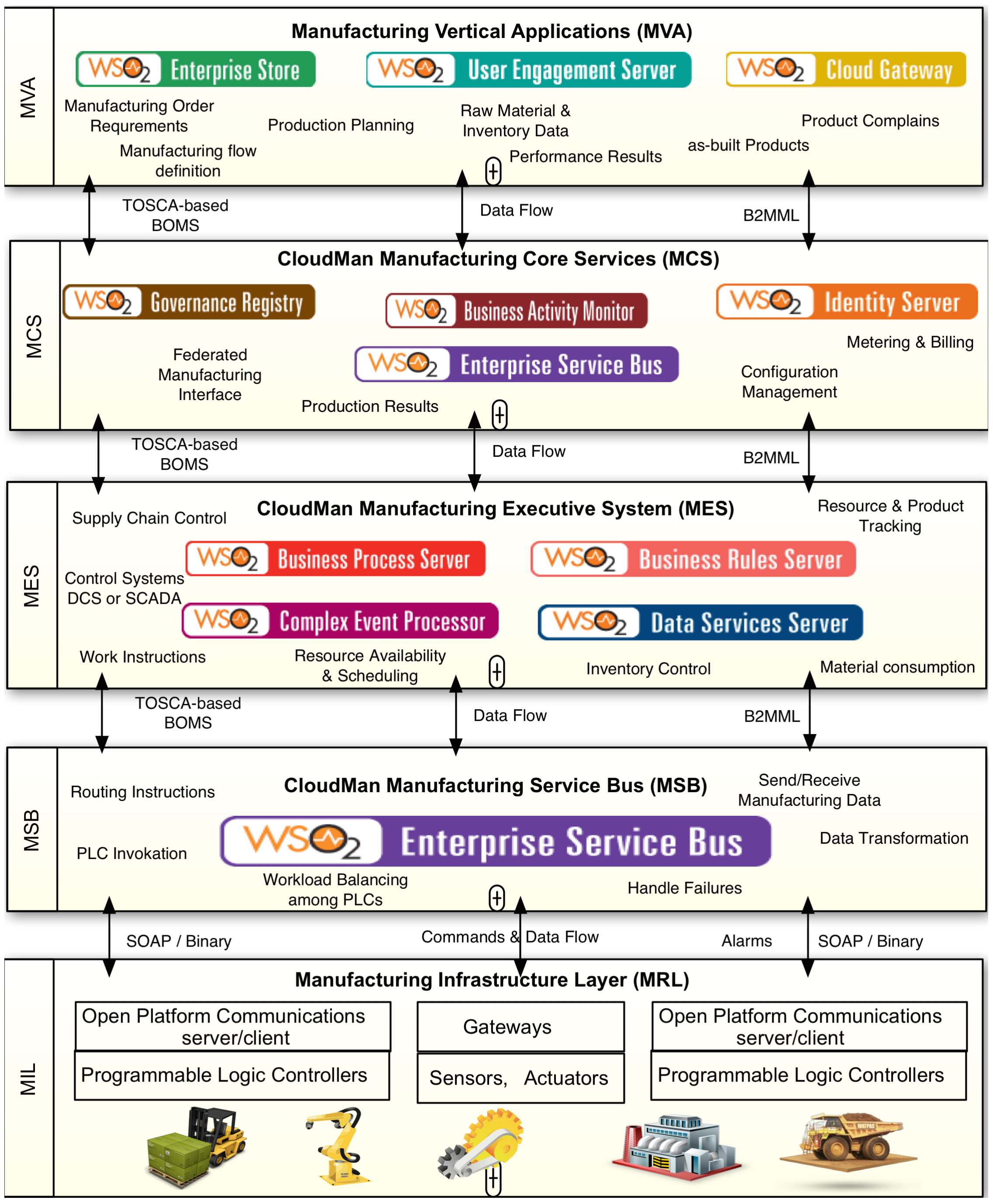
4.6.1 Layer 1. Manufacturing Virtual Applications (MVA) Layer:

At this layer, clients are involved. CloudMan provides interfaces for users to invoke the services for their business needs, like on-demand manufacturing requests. Following the request, required services will be composed and orchestrated to fulfill the order. Business users can also create web components like dashboards or gadgets to have a close look at their ongoing manufacturing orders. This layer simply represents the "*face*" of the CloudMan.

On the other side, the manufacturers can benefit of their enterprise store to advertise their manufacturing services, APIs and applications, so that the users can rapidly discover, subscribe and make use of them upon their demands. CloudMan is capable of outsourcing the manufacturing processes in an on-demand fashion using the cloud gateway service. This opens a secure channel to other manufacturing services in the cloud for more elasticity and collaboration. Last but not least, 3rd party applications like ERP/CRM can also be integrated to the CloudMan and render their manufacturing orders on cloud. They can model their requirements by product definitions, bill of materials, work instructions and the equipment settings. In return, CloudMan will respond with the manufacturing results, including the finished products, material consumptions and metering data for charging, billing and future planning purposes.

[6]The B2MML standard defines a format for exchange of ISA-95 information and the specific method (XML documents) for exchanges.

[7]http://wso2.com/products



MVA
Manufacturing Vertical Applications (MVA)
WSO2 Enterprise Store
WSO2 User Engagement Server
WSO2 Cloud Gateway
Manufacturing Order Requrements
Production Planning
Raw Material & Inventory Data
Product Complains
Manufacturing flow definition
Performance Results
as-built Products
TOSCA-based BOMS
Data Flow
B2MML
MCS
CloudMan Manufacturing Core Services (MCS)
WSO2 Governance Registry
WSO2 Business Activity Monitor
WSO2 Identity Server
Federated Manufacturing Interface
WSO2 Enterprise Service Bus
Metering & Billing
Configuration Management
Production Results
TOSCA-based BOMS
Data Flow
B2MML
MES
Supply Chain Control
CloudMan Manufacturing Executive System (MES)
Resource & Product Tracking
WSO2 Business Process Server
WSO2 Business Rules Server
Control Systems DCS or SCADA
WSO2 Complex Event Processor
WSO2 Data Services Server
Work Instructions
Resource Availability & Scheduling
Inventory Control
Material consumption
TOSCA-based BOMS
Data Flow
B2MML
MSB
Routing Instructions
CloudMan Manufacturing Service Bus (MSB)
Send/Receive Manufacturing Data
PLC Invokation
WSO2 Enterprise Service Bus
Data Transformation
Workload Balancing among PLCs
Handle Failures
SOAP / Binary
Commands & Data Flow
Alarms
SOAP / Binary
MIL
Manufacturing Infrastructure Layer (MRL)
Open Platform Communications server/client
Gateways
Open Platform Communications server/client
Programmable Logic Controllers
Sensors, Actuators
Programmable Logic Controllers

Figure 4.3: CloudMan Framework Layered Architecture

4.6.2 Layer 2. Manufacturing Core Services (MCS) Layer:

This layer is the foundation and "*body*" of the architecture, which serves all parts of the ecosystem. It relies on scalability of distributed service bus. At this layer, the Enterprise Service Bus (ESB) is in place to act as connecting layers and implement interactions among various components within layers. ESB delivers monitoring, routing and queuing of messages plus choreography and queuing of events. Another responsibility of this layer is the Identity Server, which provides the authorization service over resource usage, grants the authorized users to access the manufacturing service catalog and invoke registered services in Governance Registry.

This layer also deals with monitoring manufacturing system performance upon predefined KPIs. The monitoring mechanism consolidates the collected data around KPIs for the manufacturing system and provides insights to the stakeholders for their future planning. Monitoring is applicable on various granularity of the manufacturing units. On a higher level of granularity, the web services that are exposed for internal usage, should meet the performance criteria, like availability and response time for the prospective manufacturing load. Looking at the production level, for instance, manufacturing resources like equipments and devices (e.g., 3d Printer) running on the shop floor, need to be checked periodically to ensure their healthy operation, such as the resource status, its energy consumption and the resource utilization. These monitoring tasks might lead to a change in manufacturing flow and request the production line to elastically scale and adapt by allocating more or releasing manufacturing resources to newly detected situation.

4.6.3 Layer 3. Manufacturing Execution System (MES) Layer:

MES layer represents the "*brain*" of the CloudMan. MES receives the product definition, TOSCA-based BOMS, the manufacturing instructions, the equipments configurations and finished product quality constraints from the MVA layer or from 3rd party ERP/CRM applications. Then this layer checks the availability of manufacturing resources and does the inventory and supply chain control through the MSB layer. Having that in mind, the MES schedules the production planning and reserves the resources through Match-making APIs furnished in requirement section accordingly. The resource scheduling process deals with the distributed allocation of planned manufacturing operations associated to all orders on the available resources. This layer is also capable of defining rules and constraints for performance measurement and quality assurance.

As scheduled, the production plan will be deployed on the production lines, allocating resources and invoking manufacturing services leading to actual shop floor operations. Through Complex Event Processing (CEP), CloudMan processes events, including material consumption; resource usage and product assembly will be tracked and analyzed, then the production results will be sent to upper layers for aggregation and computation. The data services in this layer will store, retrieve and manipulate the manufacturing data to straightforward integration with manufacturing flows.

4.6.4 Layer 4. Manufacturing Service Bus (MSB) Layer:

After the production scheduling is confirmed, the execution commands and allocation messages are sent from MES layer to MSB. The MSB receives the messages/events, then after analyzing them decides which end points (e.g., machine, equipment, human) is the recipient, so that it will route the message like instructions to the desired receiver. It is in fact the "*heart*" of the CloudMan as it pumps pure manufacturing messages, data and events into various end points. MSB handles the transactions and can load balance among multiple PLCs to perform under high loads. It is also responsible for handling the production failures in case of occurrence.

4.6.5 Layer 5. Manufacturing Infrastructure (MI) Layer:

This layer encapsulates manufacturing resources like materials, machines, equipments, devices, sensors and human workforce. In CloudMan, the resources are connected, sensed and can be controlled through IoT technologies. Then these resources and their operations are virtualized and encapsulated into registered cloud manufacturing services, which will be scheduled upon the demand in MES. It is strongly the "*skeleton*" of the CloudMan as all other layers are functioning on top of this layer. Optimized resource scheduling is a key factor in this layer for increasing manufacturing productivity. Therefore there is a great need to develop a systematic real-time equipment health evaluation and dynamic preventive maintenance.

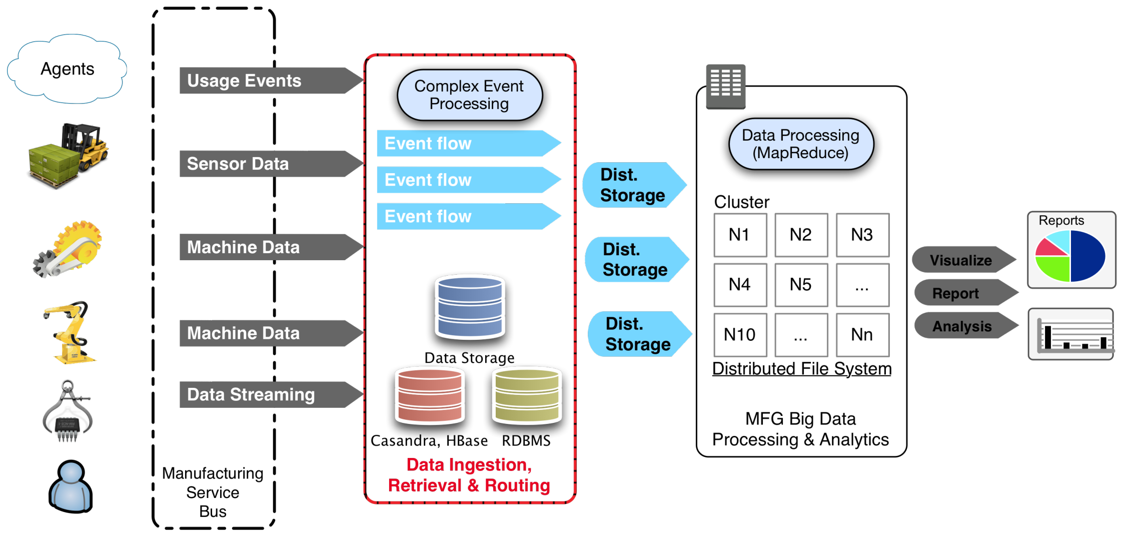


Figure 4.4: CloudMan Manufacturing Platform Big Data Architecture

4.7 CloudMan Platform Data Architecture

From the time a request is submitted to the CloudMan till its fulfillment, a huge amount of data is produced by manufacturing resources such as equipments and sensors. In order to make use of this massive volume of data that is moving from bottom most layer to the top, we need to effectively collect them, analyze them, appropriately visualize them and accordingly take actions. Here, two aspects are considerable: (i) space: as we are dealing with Peta-bytes of data from different data sources, an effective way of storing them and correlate them plays an important role, and (ii) time: in some cases we are interested in analytics results over a time period (e.g. for monitoring purposes), and in some other cases a real time analytics is needed (e.g. for generating alerts). CloudMan data architecture as shown in Figure 4.4 covers both aspects.

First stage is to collect data streams and store them in distributed large storages on HDFS (WSO2 Business Activity Monitor (BAM) is capable of doing this). Then the analysis engine such as BAM analyzer, processes the stored data and generates the results streams, which can be then stored in RDBMS databases, as the volume of analyzed data is much smaller than the original data, or we can store them again in our big data storage, if the volume is high. Our analysis engine is empowered by Hadoop framework, supporting powerful programming models such as MapReduce.

For real time analytics, we use a complex event processing engine such as WSO2 Complex Event Processing (CEP)[8] to analyze the data as they are coming and without storing them. The output stream from real time analytics generated by CEP, as well as batch analytics generated by BAM, will be sent to MVA layer to be visualized. At this layer we take benefit of tools, such as WSO2 User Engagement Server (UES)[9], to build dashboards and visualize and monitor KPIs. For the outputs from CEP, we can also generate alerts for matching event detections.

4.8 Related work

In relation to our work, there are some prominent approaches contributing architectures, platforms and models of complete frameworks for the cloud manufacturing system. Brecher et al. [33] proposed a module-based and configurable manufacturing platform based on Service-Oriented Architecture (SOA), called open Computer-Based Manufacturing (openCBM). STEP standards[10] are utilized to preserve the results of manufacturing processes that are fed back to the process planning stage. Li et al. [34] introduced a four-layer application service integration platform that is able to bridge multiple clouds and information systems. Interactions across organization boundaries are supported by collaboration point, which plays as an interface providing data exchange, command transferring, monitoring and so forth. The system integrated manufacturing business processes

[8]http://wso2.com/products/complex-event-processor

[9]http://wso2.com/landing/user-engagement-server

[10]http://en.wikipedia.org/wiki/ISO_10303

with the help of collaboration agents. This research work examined the possibility of integrating existing manufacturing applications in the cloud Computing environment. Schulte at el.[35],[36] analyzes requirements regarding process enactment for Cloud manufacturing and provides a concept for an according software framework. They also elaborate the use of service-oriented virtual factories to establish, manage, monitor, and adapt virtual factories in a plug-and-play-like fashion.

Wang and Xu [37] proposed a Distributed Inter-operable Manufacturing Platform (DIMP) as an integrative environment among existing and future CAD/CAM/CNC applications. It is also based on SOA concept. In the platform, the requests and tasks from the users are modeled, collected and defined as "Virtual Service Combination", which echoes the manufacturing requirement at the Global Service Layer. Moreover, STEP data models are utilized as the central data schema. In a recent paper[38], they have proposed a service-oriented system called Interoperable cloud-based Manufacturing System (ICMS) which provides a cloud-based environment, integrating the existing and future manufacturing resources by packaging them using the "Virtual Function Block" mechanism and standardized description.

In contrast to existing approaches, our CloudMan platform is incorporating a cloud standard to elicit the manufacturing requirements using the TOSCA-based Bill of Manufacturing Services (BOMS).

4.9 Conclusion

The cloud-based design and manufacturing refers to a product realization model under the support of three core technologies of Cloud computing, Internet of Things (IoT) and Collaborative engineering. In this chapter, we proposed a reconfigurable cloud manufacturing platform, called CloudMan platform, which considers these three technologies and deals with the portability and provisioning of all types of manufacturing resources as services. We incorporate TOSCA specification to model product Bill of Materials(BOM) as Bill of Manufacturing Services (BOMS) which leads to portability of such services. This elevates the manufacturing process to be more resistance against failure and defects detected in one production line since the manufacturing service can be ported to other certified production eventually. As an outlook, our future work will more focus on utilizing TOSCA-based BOMS specification in realization of virtual factories which are formed and constructed on-demand by a dynamic composition of distributed manufacturing services and resources with the CloudMan platform.

CHAPTER 5

Linked Sensor Data

5.1 Introduction

A data stream is a massive sequence of sensed data. Sensing is a process of expressing true values in context. Context refers to an environment of interest in which a sensor is embedded. Such sensors might be a device or a service. They represent the context behavior by streaming the values of its computational objects attributes. To interpret the situation of a thing in a context, semantics of such raw data seems to be vital. An Internet of Things (IoT) incorporates cloud computing[3] and virtualization mechanisms to expose such data to analytic endpoints to drive context awareness. The W3C Semantic Sensor Network Incubator Group (SSN-XG)[1] has developed an ontology to elevate the quality of sensed data with semantic web technologies. Project Haystack[2] has developed naming conventions and tags for environmental equipment like buildings and lighting devices together with their operational data. Having these conventions in place, clients are able to consume Haystack REST APIs to discover, query, and tag objects and the data collected and stored by IoT frameworks like NiagaraAX[3]. IoT infrastructure is composed of resource-constrained gateways and a network of devices. The above-mentioned solutions are not lightweight and universal in terms of their provisioning and deployment model. They also lack analytic endpoints for reasoning purposes.

To this end, our contribution is threefold: (i) A built-in solution for the semantic sensor data retrieval on the IoT device layer. We develop a semantic gateway middleware called Gatica, which delivers automated and on-demand semantic annotations, labels and taxonomies for sensor data acquisition at scale. (ii) In support of such linked sensor data which was collected and annotated by the wrappers, we provide a "manifest" as the meta-data dictionary of the sensor current readings. This contributes to real-time

[1]http://www.w3.org/2005/Incubator/ssn/
[2]http://project-haystack.org
[3]http://www.niagaraax.com

semantic sensor data retrieval. (iii) At streaming-time, the sensor annotated data object from the wrapper is injected into the mediator and the mediator virtually applies in lightweight transformations on raw data resulting RDF triples. Our Gatica middleware implements a layered approach to the interpretation of the sensor time-series data. The linked data RDFs are then streamed to the analytics endpoints for querying and reasoning purposes.

This chapter continues with an initial analysis over the utility of the data source being studied in this research in section 5.2. After the data utility mathematical model is detailed, section5.3 presents the basic concepts and preliminaries of IoT gateways. With some definitive clues on data source structure and its associated annotations, we propose a novel IoT gateway middleware to fulfill sensor data enrichment. Section 5.4 is devoted to the core elements of Gatica layered architecture together with its interacting components. In support of our model, we deploy our middleware to production by processing the real-world medical data set. Subsequently, section 5.5 surveys related work. Finally, section 5.6 concludes the chapter and presents an outlook on future research directions.

5.2 The Utility of Sensed Data

There is a commendable survey[39] in which the authors explore the state of the art of how the health-care sensed data are utilized by applying analytics and mining algorithms. Along with such data use-cases, patients requiring intensive care need continuous observation and treatment. This is achieved via wearable or contextual sensors, which are connected to medical devices measuring physical attributes and producing a considerable amount of vital data on a per-patient basis. Medical institutions that are collecting big amounts of such data enable us to achieve a better understanding of the patient's current status and their recovery progress.

Having such data in place, the health-care service providers are able to construct a wellness-function for the *normal* range of the vitals and produce alerts upon on deviating from the normal values. Through this vital range interpretation, various disease patterns can be discovered together with its severity.

In this chapter we have used the Massachusetts General Hospital/Marquette Foundation (MGH/MF) Waveform Database[40] that represents a comprehensive collection of electronic recordings of hemodynamic and electrocardiographic waveforms of stable and unstable patients in critical care units, operating rooms, and cardiac categorization laboratories.

This data set is used as a real world motivation scenario in putting Gatica in production mode. At any given time t, Gatica provides enriched pieces of information about the medical sensor observations. Such observations can be represented as a column-vector $\mathbf{o}_t \equiv [v_{t,1}\ v_{t,2}\ \ldots\ v_{t,n}]^T \in R^n$ of sensor data stream values at time t. The stream data can be regarded as a frequently expanding $t \times n$ matrix $\mathbf{O}_t := [\mathbf{o}_1\ \mathbf{o}_2\ \ldots\ \mathbf{o}_t]^T \in R^{t \times n}$ where the new incoming streams are added as matrix rows at each time interval t in real-time. In our health-care example, $\mathbf{O}_t$ is the measurements column-vector at t over all the sensors, where n is the length of the vector and indicates the number of health-care sensors and t is the measurement time-stamp. These vectors represent the set of measurements obtained by the n sensor at a specific observation. In particular the rows of the matrix represent the different observations in a given period, while the columns the sample values detected from each sensor during the observations.

In our scenario, as shown in Equation 5.1, let $\mathbf{O}$ be a matrix representing the patient vital data measured by sensors in the hospital ICU room. For instance, vector ECG_i represents the electro activity of heart beats, vector ART_j observes the blood pressures and vector $CO2_k$ indicates the levels of blood carbon dioxide in a period of time.

$$\mathbf{O}_{t,n} = \begin{pmatrix} ECG_{1,1} & ART_{1,2} & \cdots & CO2_{1,n} \\ ECG_{2,1} & ART_{2,2} & \cdots & CO2_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ ECG_{t,1} & ART_{t,2} & \cdots & CO2_{t,n} \end{pmatrix} \tag{5.1}$$

Then we perform Principal Component Analysis (PCA)[41], $PCA(\mathbf{O}) \rightarrow \mathbf{O}'$, which divides the matrix $\mathbf{O}$ into components to extract the patients health behavior pattern. The PCA orthogonalizes the columns (take set of orthogonal vectors) of $\mathbf{O}_{t,n}$ with the *Gram-schmidt*[4] process as shown in Equations 5.2 and 5.3.

$$\mathbf{v}_1 = \mathbf{w}_1 = \begin{vmatrix} ECG_{1,1} \\ ECG_{2,1} \\ \vdots \\ ECG_{t,1} \end{vmatrix} \cdots \mathbf{w}_2 = \begin{vmatrix} ART_{1,2} \\ ART_{2,2} \\ \vdots \\ ART_{t,2} \end{vmatrix} \tag{5.2}$$

$$\mathbf{v}_2 = \mathbf{w}_2 - \frac{\langle v_1, w_2 \rangle}{\langle v_1, v_1 \rangle} v_1 \tag{5.3}$$

where $\langle v, w \rangle$ denotes the inner product of the vectors v and w. This recursive process generates the set of orthogonal principal vectors as generalized in Equation 5.4.

$$\mathbf{v}_n = \mathbf{w}_n - \sum_{i=1}^{n-1} \frac{\langle v_i, w_n \rangle}{\langle v_i, v_i \rangle} v_i \tag{5.4}$$

The result is matrix $\mathbf{O}'$ with orthogonal components with minimized dimension of f.

[4] http://en.wikipedia.org/wiki/Gram–Schmidt_process

$$\mathbf{O'} = \begin{vmatrix} ECG'_{1,1} \\ ECG'_{2,1} \\ \vdots \\ ECG'_{t,1} \end{vmatrix} \begin{vmatrix} ART'_{1,2} \\ ART'_{2,2} \\ \vdots \\ ART'_{t,2} \end{vmatrix} \cdots \begin{vmatrix} CO2'_{1,f} \\ CO2'_{2,f} \\ \vdots \\ CO2'_{t,f} \end{vmatrix} \texttt{with } f \preceq n \tag{5.5}$$

After performing the PCA, we will have a set of components/vectors where each one represents observations of one header in the data set.

These extracted vectors together with the patients profile PP vector are then modelled as a matrix $\mathbf{D}$ in the Equation 5.6. This matrix will be used to detect and discover the *diagnoses* via some correlation pattern discovery.

$$\mathbf{D} = \begin{vmatrix} PP_{1,1} \\ PP_{2,1} \\ \vdots \\ PP_{t,1} \end{vmatrix} \begin{vmatrix} ECG'_{1,1} \\ ECG'_{2,1} \\ \vdots \\ ECG'_{t,1} \end{vmatrix} \begin{vmatrix} ART'_{1,2} \\ ART'_{2,2} \\ \vdots \\ ART'_{t,2} \end{vmatrix} \cdots \begin{vmatrix} CO2'_{1,f} \\ CO2'_{2,f} \\ \vdots \\ CO2'_{t,f} \end{vmatrix} \tag{5.6}$$

Using the above equation, we compute a correlation matrix $\mathbf{C}_{x,y}$ where x and y represent matrix indices to find the relationships among variables. The $\mathbf{C}_{x,y}$ is given by the correlation coefficient[5] $c_{x,y}$ between the $\mathbf{D}_x$ and $\mathbf{D}_y$.

$$\mathbf{C}_{x,y} = Corr(\mathbf{D}_x, \mathbf{D}_y) = \frac{\sigma_{xy}}{\sigma_x \sigma_y} \tag{5.7}$$

The Equation 5.7 contains terms of correlation where σ_{xy} indicates the covariance between $\mathbf{D}_x$ and $\mathbf{D}_y$. The two variables of σ_x and σ_y represent the standard deviation of $\mathbf{D}_x$ and $\mathbf{D}_y$. This constructs various sub-matrices acquiring and utilizing cross-correlation patterns of actual health status observations over patient's hospitalization profile within a specific intensive care period. Such discovered patterns can be clustered using K-means[6] or Bond Energy Algorithm (BEA)[42] methods, for instance, into a similar behavior patterns to diagnose the disease based on their similarities to the centroids of the cluster. These centroids of our k-means can be defined as the attributes of a specific disease.

The major issue with this raw sensed data is the lack of semantics to detect early diagnosis. The interpretation of the sensor data into meaningful prescription requires a deep understanding of the medical information and should be driven by domain experts. Since the data is raw, the medical sensor records should be linked with the labels to ease the disease detection by doctors. This leads to improved delivery of care by providing the health caring services to patients in an proactive fashion. With this motivation in mind, we proceed with some IoT gateway preliminary concepts.

[5] Pearson's correlation coefficient between two variables is defined as the covariance of the two variables divided by the product of their standard deviations.

[6] http://en.wikipedia.org/wiki/K-means_clustering

5.3 IoT Gateways: Terms & Preliminaries

IoT gateways are resource-constraint devices (i.e, with limited compute, memory, and storage amounts) which expose connected sensors as cloud services to become addressable, discoverable and controllable. In order to operate our gateway, we use the size-optimized and tailored BusyBox[7] OS which is a combination of UNIX utilities into a single small executable. On top of the OS, a provisioning framework is required to deploy and manage the life-cycle of the lightweight execution units or IoT tiny applications. In our architecture, we used the Sedona[8] framework for such large-scale application deployments in very constrained embedded environments.

In our architecture, we have utilized the Sedona framework in the medical gateway to build a solution for aggregating device signals and performing some operations through lightweight execution units (Sedona apps) which have been deployed on each Gateway. Now, we elaborate a bit more into Sedona core concepts:

◇ *Classes*: they extend the *Component* class to perform the defined tasks. The *component* class includes *Slots* that specify how the component is exposed.

◇ *Kits*: Sedona language sources are compiled into fine-grained modularity archive files with the ".kit" extension. Kits include application's *Manifest* and *Intermediate Representation (IR)* file that is a non-executable compiled *Component* class in Assembly. All kits are compiled into a compact *SCode* binary image which will be executable on Sedona VM.

◇ *Manifest*: Each Sedona application has a manifest file which contains meta-data of the *Kit* like name, version, build host and dependencies.

◇ *Sedona Virtual Machine*: The SVM is a small interpreter written in C designed for portability. It allows *Kits* to be executed on any Sedona-enabled device.

Sedona is a component oriented language, so composing the application by assembling pre-defined components is a principle in this framework. By assembling the required components and compiling it, there will be a file with ".sab" extension. This file is deployable on the Sedona device what we call it lightweight execution unit in our gateway. In real world, there are different Sedona Gateway devices with different specifications, for instance, Raspberry Pi[9] device including Busybox linux 2.6.32 kernel. To simulate this device as our gateway, we used BusyBox hosting the Sedona VM. This image is running on a docker[10] container to mimic the physical gateway. Once the SVM is up and running, the device will be discoverable from our client. Next, we detail the Gatica's architecture in the following section.

[7]http://www.busybox.net
[8]http://www.sedonadev.org
[9]http://en.wikipedia.org/wiki/Raspberry_Pi
[10]https://www.docker.com

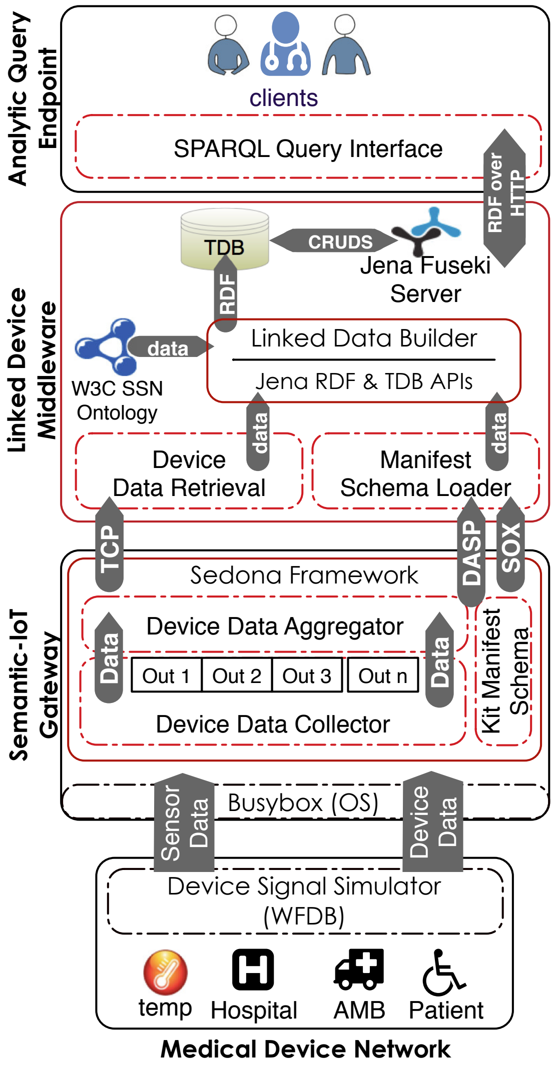
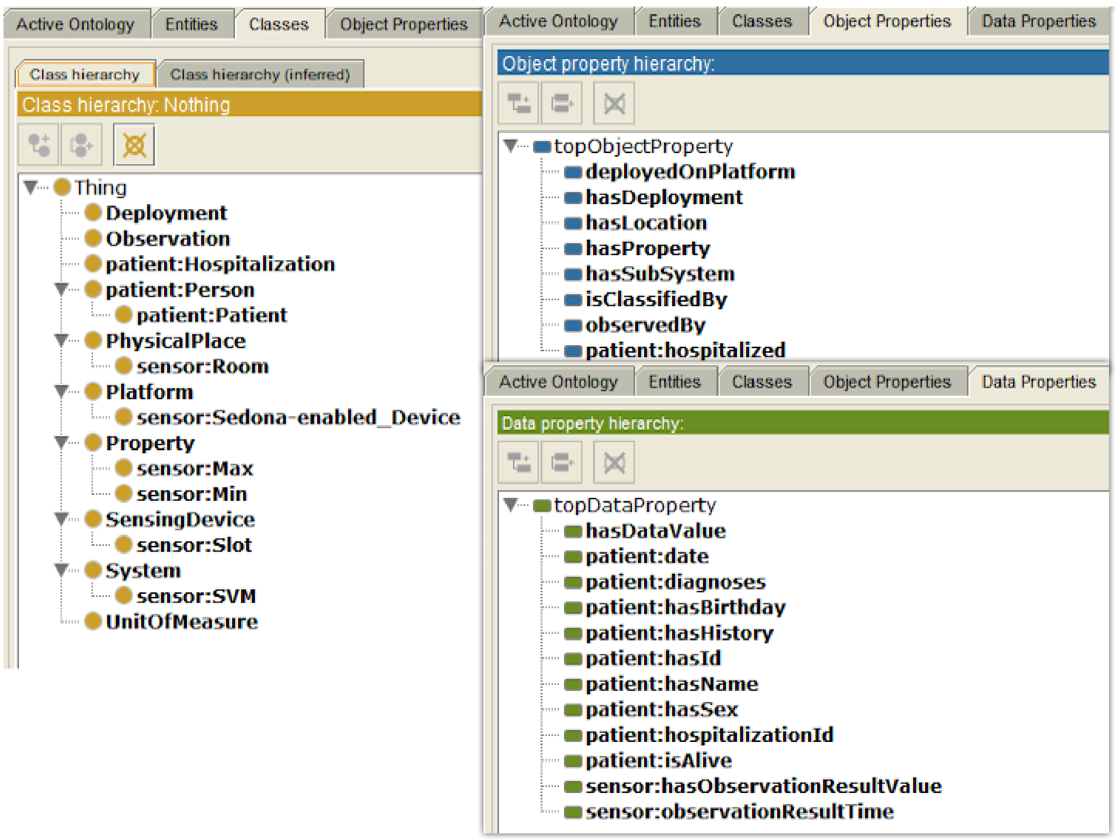


Figure 5.1: Gatica Middleware layered architecture

5.4 Gatica Middleware Architecture

Looking forward, Figure 5.1. illustrates a schematic view on architecting Gatica's collaborating components. As a blueprint for designing the architecture, the Gatica is organized into four interconnected layers: (i) Medical Device Network, (ii) Semantic IoT Gateway, (iii) Linked Device Middleware and (iv) Analytic Query Endpoint. Here the whole architecture is implemented[11]. Next each layer's specification and capabilities together with its implementation details are described.

[11]https://github.com/soheil4TUWien/Gatica



Class hierarchy: Nothing

- Thing
  - Deployment
  - Observation
  - patient:Hospitalization
  - patient:Person
    - patient:Patient
  - PhysicalPlace
    - sensor:Room
  - Platform
    - sensor:Sedona-enabled_Device
  - Property
    - sensor:Max
    - sensor:Min
  - SensingDevice
    - sensor:Slot
  - System
    - sensor:SVM
  - UnitOfMeasure

Object property hierarchy:

- topObjectProperty
  - deployedOnPlatform
  - hasDeployment
  - hasLocation
  - hasProperty
  - hasSubSystem
  - isClassifiedBy
  - observedBy
  - patient:hospitalized

Data property hierarchy:

- topDataProperty
  - hasDataValue
  - patient:date
  - patient:diagnoses
  - patient:hasBirthday
  - patient:hasHistory
  - patient:hasId
  - patient:hasName
  - patient:hasSex
  - patient:hospitalizationId
  - patient:isAlive
  - sensor:hasObservationResultValue
  - sensor:observationResultTime

Figure 5.2: Sensor ontology classes, objects and data properties

5.4.1 Sensor Data Retrieval

Now, we briefly review the data flow from the sensor/device network to the upper layers. Gatica is designed to cope with large amounts of real-time sensor data by transforming it into linked data. This includes operations involved in collecting data from external sensor data sources. Then, preprocessing operations like cleansing noisy data are applied to the data to prepare it for further analysis. The sensor data acquisition is performed by the deployed kits in the device layer that interfaces with sensors and feeds into the stream processing system. This data is a time series consisting of ordered sequences of `[key,value]` pairs of timestamps and data elements.

In the MGH/MF Waveform Database, three files describe each record. These files are ".ari" extension (beat and event annotation), ".dat" extension (digitalized signal (s)) and ".hea" extension (header file). In Device Signal Simulator, we have used the *mgh001*[12] waveform database. The headers for this waveform record include the following nine elements: 1. Timestamp 2. ECG lead I, 3. ECG lead II, 4. ECG lead V, 5. ART, 6. PAP, 7. CVP, 8. Resp. Imp, and 9. CO2 which are described in the header file of the data

[12]http://physionet.org/physiobank/database/mghdb/mgh001.hea

set. We have implemented a C program which simulates the sensor behavior by reading signal data from *mgh001*. In effect, this utility iterates over total records of database and will send each record including nine fields over a TCP socket connection to DDC. DDC will receive each record and tokenize the `value` into eight five-bits signal digits.

Each running SVM which is located in an ICU room has a unique deployment ID. This enables us to extract information for each SVM such as its IP address, location and the hospitalized patient ID. In our implementation we have defined a SVM_ID variable in DDA kit. At this moment we have assigned a default value to this variable to run the prototype. DDA aggregates sensor data and sub-joins the SVM_ID to the aggregated data message, then sends it to the LDM. But in a real world application, each SVM will ask the deployment server a unique ID at startup. Then the server sends a unique ID (SVM unique ID + patient ID) to the SVM device. This ID will be used in DDA kit.

In the Sedona framework, we implemented *Device Data Collector (DDC)* and *Device Data Aggregator (DDA)* applications which receives sensor data through its listeners via TCP Socket and aggregates the data for further processing. More specifically, DDA kit receives data from proper input slots which are linked to the outputs of DDC kit. The output-input link tags are defined in DDA sax file. This kit merges all inputs separated by semicolon and sends the aggregated byte stream via a TCP Socket object on an IP and port number which our next component, *Device Data Retrieval (DDR)* is hosted.

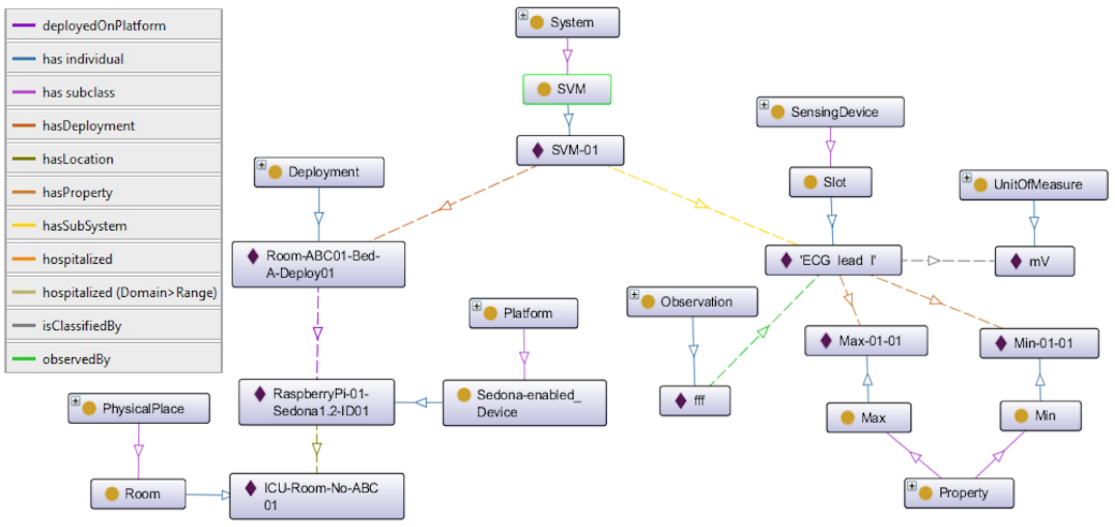


Figure 5.3: Sensor ontology model representing the classes, relations and instances

5.4.2 Artefact Manifest Schema

In our aggregator application, we define properties of WFDB fields as the meta-data for this SVM environment. These properties will be stored in the *manifest* XML file in compile-time. Then we use Sox protocol to retrieve each property of the manifest using the *Manifest Schema Loader (MSL)* in the next layer. To communicate with our Sedona device remotely and retrieve information about running Sedona VM, we used Datagram Authentication Session Protocol (DASP)[13]. In effect, we used DASP Socket in our middleware to send a multi-cast group request (DISCOVER request). Therefore each machine running Sedona VM (as server) will respond to the client. Notice that the discovery request will query a specific port address on destination machine which in our case is Sedona default port number 1876. As soon as each machine receive the discover request, it will response with the Platform ID and its IP address. In our case the DASP message received from server contain following SVM information: `Discovered SVM IP/Port: 192.168.1.3:1876` and `Platform ID: tridium-generic-unix-1.2.28`. Such information has two usecases of (i) sending commands to control the device from the analytic endpoint side and, (ii) as a heartbeat to check if the running SVMs are still alive and healthy.

5.4.3 Sensor Linked Data Model

The detected events, which in our particular case are user health vital information, can be modelled in an ontology. The ontology model as depicted in 5.3, consists of two parts of sensor and patient parts. The patient part represents her personal and medical profile like hospitalization ID and the detected diagnosis, for instance. Every profile is linked via hospitalization ID to deployment ID in sensor ontology.

Sensor ontology constitute a network of sensors which are connected to a Sedona device and deployed in a Sedona Virtual Machine (SVM). Such sensors read vital data from the patient. For modeling sensor properties, capabilities and their observations we incorporated the W3C SSN ontology[14]. We utilize this ontology by representing our located Sedona device instance. The Sedona VM is mapped to the *system* class which is deployed on a platform with a specific deployment ID. The sensors are considered as *subsystems* and declared as *slots*. Slots have properties like ID, label, measurement unit together with their ranges like min or max values. They expose their located context observation results.

As shown in Figure 5.2. in addition to using object and data properties of SSN ontology, we defined *hasID*, *hasObservationResultValue* and *observationResultTime* properties. The *hasId* is defined for assigning the related IDs to SVM, slot and deployment. As such, the *hasObservationResultValue* is defined for assigning sensor readings value to observation of each slot. And finally, the *observationResultTime* for assigning an integer as a timestamp of reading observation of each slot. The ontology is built and loaded into the TDB RDF database[15] using the Jena TDB APIs to store and retrieve semantic sensor data in RDF graphs. We developed our sensor ontology in Protege[16]. Next, the manifest schema and observed values are written to TDB. We have used its APIs to store and expose

[13]http://www.sedonadev.org/doc/dasp.html
[14]http://www.w3.org/2005/Incubator/ssn/ssnx/ssn
[15]https://jena.apache.org/documentation/tdb
[16]http://protege.stanford.edu

our enriched sensor data as Triple RDF statements. Meanwhile some sample patient hospitilization data is stored in the database for future retrieval.

5.4.4 Manifest Schema Loader

We need to fetch the meta data of the deployed sensors (slots) on the Sedona device. The manifest file contains annotation data related to each slot such as min, max or a measurement unit. We use Sedona APIs to read such data that exists in the manifest file. A simple sox client is implemented which communicates, authenticates and listens on a sox port. This thin client requests slot information like names, values, flags, and annotations from the manifest schema. The response holds the manifest data which will be written to the TDB afterwards.

5.4.5 Device Data Retrieval

Next is to store the actual sensor observation values in the DB. As described before, from each sensor the stream of data (observations) are loaded in the Sedona device. Our Sedona main application merges the SVM ID, slot ID, observed value and its time-stamp, then sends them via TCP to a specific port. At last, the DDR receives the data periodically and stores them in the TDB.

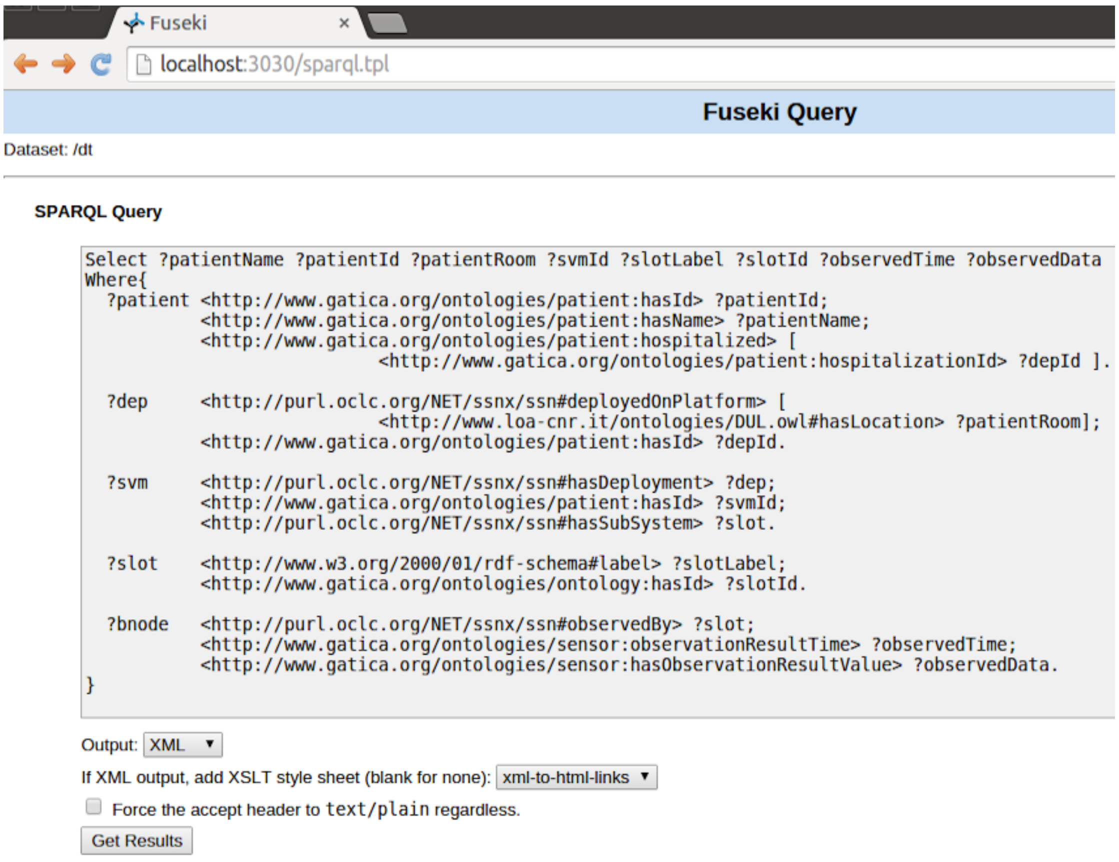
5.4.6 Analytic Query Endpoint

As shown in Figure 5.4. we implement SPARQL querying interface on the TDB by using Fuseki[17] to serve RDF data over HTTP. After the Fuseki server is running, a SPARQL endpoint is provided to respond to various SPARQL requests. A sample response is presented in Figure 5.5. Various analytic applications can send SPARQL request to our Fuseki server and retrieve the results.

5.5 Related Work

There is some valuable research regarding the IoT semantic sensor streaming which elevates the semantic technologies to Internet of Things domain. Banerjee et al.[43] proposed an architecture for a semantic search engine on sensor data where the sensed data is represented in form of triples (RDF), concepts and relations in form of ontologies (OWL) and the corresponding query language is SPARQL. In relation to our approach, Le-Phuoc et al.[44], [45] survey the state of the art Linked Stream Data processing systems, and highlights a comparison among them regarding the design choices. Authors in [46], propose a model and take a linked data approach to annotate the streams. These papers do not provide any implementation or evaluation of the proposed architecture. They also proposed a Linked Sensor Middleware (LSM) [47] that provides real-time data collection mechanisms using a cloud-based infrastructure. It also features a web interface

[17]http://jena.apache.org/documentation/serving_data



Dataset: /dt

SPARQL Query

```
Select ?patientName ?patientId ?patientRoom ?svmId ?slotLabel ?slotId ?observedTime ?observedData
Where{
  ?patient <http://www.gatica.org/ontologies/patient:hasId> ?patientId;
           <http://www.gatica.org/ontologies/patient:hasName> ?patientName;
           <http://www.gatica.org/ontologies/patient:hospitalized> [
                    <http://www.gatica.org/ontologies/patient:hospitalizationId> ?depId ].

  ?dep     <http://purl.oclc.org/NET/ssnx/ssn#deployedOnPlatform> [
                    <http://www.loa-cnr.it/ontologies/DUL.owl#hasLocation> ?patientRoom];
           <http://www.gatica.org/ontologies/patient:hasId> ?depId.

  ?svm     <http://purl.oclc.org/NET/ssnx/ssn#hasDeployment> ?dep;
           <http://www.gatica.org/ontologies/patient:hasId> ?svmId;
           <http://purl.oclc.org/NET/ssnx/ssn#hasSubSystem> ?slot.

  ?slot    <http://www.w3.org/2000/01/rdf-schema#label> ?slotLabel;
           <http://www.gatica.org/ontologies/ontology:hasId> ?slotId.

  ?bnode   <http://purl.oclc.org/NET/ssnx/ssn#observedBy> ?slot;
           <http://www.gatica.org/ontologies/sensor:observationResultTime> ?observedTime;
           <http://www.gatica.org/ontologies/sensor:hasObservationResultValue> ?observedData.
}
```

Output: XML

If XML output, add XSLT style sheet (blank for none): xml-to-html-links

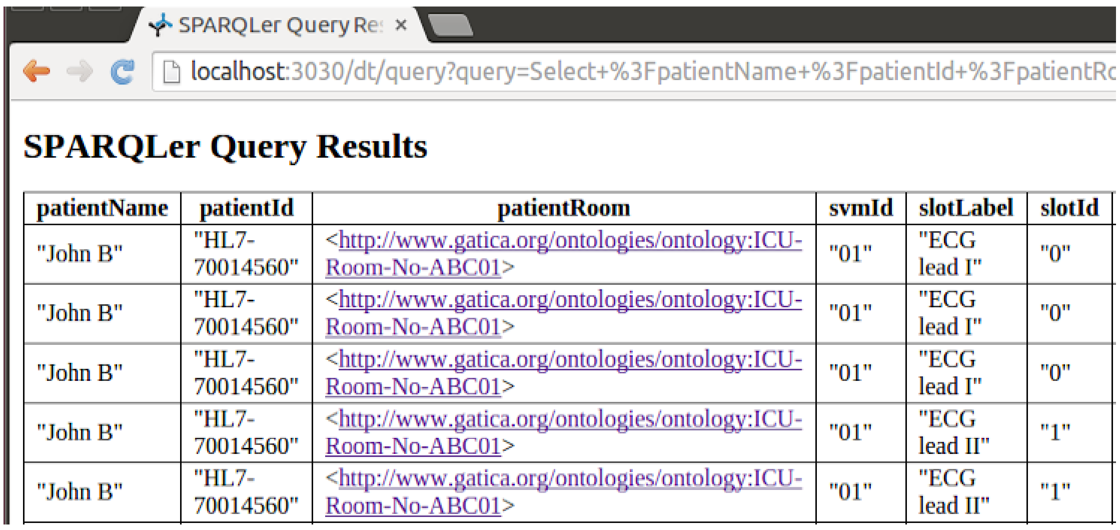
Force the accept header to `text/plain` regardless.

Get Results

Figure 5.4: Gatica middleware analytic query endpoint

for annotating and visualizing linked sensor data accompanied with a SPARQL endpoint for querying streaming data. Their focus is on annotating and provisioning of the data using unified interfaces.

None of these solutions enable a lightweight IoT deployment model and is not applicable to resource-constraint IoT devices like an IoT gateway environment. In contrast to all the noted related work, our solution addresses universal and large-scale application deployments in very constrained embedded environments. The Sedona framework is deployable on very resource limited devices with small memory, even less than 100 KB. Our solution is also based on a micro-service architecture as each Sedona application is highly coherent and decoupled from others. Last but not least, each container in the gateway is accompanied with the manifest schema which enables dynamic configurations of the annotations.


SPARQLer Query Re ×

localhost:3030/dt/query?query=Select+%3FpatientName+%3FpatientId+%3FpatientRo

SPARQLer Query Results

patientName	patientId	patientRoom	svmId	slotLabel	slotId
"John B"	"HL7-70014560"	<http://www.gatica.org/ontologies/ontology:ICU-Room-No-ABC01>	"01"	"ECG lead I"	"0"
"John B"	"HL7-70014560"	<http://www.gatica.org/ontologies/ontology:ICU-Room-No-ABC01>	"01"	"ECG lead I"	"0"
"John B"	"HL7-70014560"	<http://www.gatica.org/ontologies/ontology:ICU-Room-No-ABC01>	"01"	"ECG lead I"	"0"
"John B"	"HL7-70014560"	<http://www.gatica.org/ontologies/ontology:ICU-Room-No-ABC01>	"01"	"ECG lead II"	"1"
"John B"	"HL7-70014560"	<http://www.gatica.org/ontologies/ontology:ICU-Room-No-ABC01>	"01"	"ECG lead II"	"1"

Figure 5.5: Sample response from the query interface

5.6 Conclusion

In this chapter, the authors offered and implemented a resource-constrained middleware to enable the semantic IoT linked data analytics. It dynamically injects semantics to make the IoT raw sensor data enriched and meaningful. For modeling a sensor's properties, capabilities and their observations, we extended the SSN ontology which represents our gateway containers. Gatica collects the real-time sensor data via gateways, enrich them using annotations then transforms and exposes them in RDF triples. Using principle component analysis we are able to cluster the data and run queries over the streaming sensor data to discover hidden patterns via analytic interfaces. We have evaluated our model with a real-world healthcare dataset to demonstrate the utility of our framework.

As an outlook, our future work includes further extension to the Gatica middleware to support large scale distribution of data processing capabilities over the increasing streaming linked sensor data.

Part II

IoT Design Patterns

CHAPTER 6

IoT Computational Constructs

6.1 Introduction

Design is the use of scientific principles, technical information and imagination in the definition of a structure, machine or system to perform pre-specified functions with the maximum economy and efficiency[48]. The Internet of Things (IoT) means that hardware and software design practices blend into each other. A well-designed IoT application may be composed of utilized edge devices, fine-grained microservices, cloud gateways to connect edge network to the Internet and mobile or web applications for the user to interact with the underlying things.

There are huge opportunities but considerable challenges [49, 50, 51, 52, 53] in designing IoT applications. These challenges range from the provisioning of ultra-low power operation and system design using modular, composable components to smart automation. Furthermore, the advancement in sensor instrumentation requires an efficient stream data processing. There are also hidden opportunities and challenges in monetizing the edge applications. In response to such challenges, we propose a set of reusable and abstract prescriptive design principles or patterns, which aid system architects in modeling and building *context-tailored* IoT applications. The patterns behold an integrated thinking across many facets of design in an IoT application ecosystem by incorporating the wiring approach of Hanmer [54]. This articulates the benefits of applying patterns by showing how each piece can fit into one integrated solution.

With this motivation in mind, the chapter continues in section 6.2, with a brief review of how IoT design patterns are documented. Next, we introduce the diversity of our proposed patterns and how they are related to the edge applications life-cycle in section 6.3. With some definitive clues on the pattern language convention, we propose an *edge provisioning* pattern in section 6.4, showing how the baseline environment provisioning can be automated. Once the baseline container is provisioned, we demonstrate how code can be automatically pushed to the container via a deployment pipeline. The *code deployment* pattern interacting entities are detailed in section 6.5. Here, we also define how dynamic configuration effects the quality and performance of the service delivery. Next, in section 6.6, the orchestration of IoT services, their dependencies and configurations are focused on and presented as an *edge orchestration* pattern. Now, the composite IoT application is running and its components are willing to communicate with each other via messaging. At section 6.7, we enable edge applications to share data and functionality via a *footprint messaging* pattern. The running edge services retrieves the sensor data and in *in-memory data retrieval* pattern, we define how such data can be stored and retrieved in section 6.8. Faced with such data, we express lightweight data cleansing and validation tasks as an in-device data preprocessing pattern in section 6.9. This leads to rendering the edge application by metering its usage along with its underlying resource usage. Such a metering model is proposed as a *Diameter of Things* pattern and is detailed in section 6.10. Last but not least, we show how such edge applications function on wearable technologies using the *wearable façade* pattern in section 6.11. Subsequently, section 6.12 surveys related work. Finally, section 6.13 concludes the chapter and presents an outlook on future research directions.

6.2 Pattern Language Conventions

Pattern language is intended to describe the solution in a way that is easy to digest. We are incorporating the cloud computing pattern document format defined by Wellhausen[55] and Meszaros[56] as well as the semantics of its graphical elements to define the structure of our IoT design patterns and their interrelations. All IoT patterns comprise the same document sections with the following semantics.

☞ *Pattern Name*: This name is a handle used to abstract and identify a design challenge.

☞ *Problem*: This is a short summary of the pattern, i.e., the driving question in one or two sentences.

☞ *Context*: This section of the pattern documentation describes the setting in which the problem arises. Assessing the pattern's context influences the solution.

☞ *Motivation Forces*: This basically provides a use-case scenario, in which the problem is likely to happen and therefore the pattern can be used effectively.

☞ *Solution Details*: This section briefly states how the pattern solves the problem.

☞ *Sketch*: It depicts the functionality of the solution or the resulting architecture after application of the pattern.

We have also ensured that the pattern names are abstract-enough to reflect the pattern's intended design. This conforms to the principle of pattern's name cohesion defined by Hanmer[57].

6.3 IoT Design Patterns

Designing for an IoT is different. Connected devices may use different types of networks and various connectivity patterns. IoT design patterns vary in their granularity and level of abstraction. To create a valuable, appealing, usable, and coherent edge applications, we have to consider design on many different layers. This is a fine-grained design, as it exposes low-level data from the device itself. While documenting the patterns, we also give an overview of existing implemented frameworks to give a clue and a closer touch on the efficiency of our proposed patterns in production.

Methods for automated provisioning, deployment and configuration management of the behavior of edge applications disclosed herein pertain to governance patterns. Setting up the system and getting the devices connected are hard to simplify. IoT governance patterns deal with governing the edge applications life-cycle from definition through deployment. Such patterns govern all aspects of edge applications including their provisioning and deployment mechanisms by applying runtime reconfiguration and allocation and by scheduling policies to deployed edge services.

Having a reliable and efficient provisioning pattern in place, we proceed with communication patterns. Such patterns establish a trusted edge service bus among devices so that they share data, messages and trigger functionality. This utilizes the whole IoT ecosystem as an integrated scalable solution by bridging heterogeneous edge service units and nodes. Next, we will delve into an in-device data storage and processing. To achieve this, we introduce a pipelined framework for realtime validation and cleaning of sensed data streams.

Last but not least, we will demonstrate how edge applications are considered value-added and metered services. The proposed metering pattern implements a real-time metering of IoT services for prepaid as well as Pay-per-use economic models.

6.4 Edge Provisioning Pattern

♣ *Problem*: How can operation managers and developers ensure all of their edge devices are started with a reliable baseline environment, as needed? How can they provision all the devices automatically all at once?

♣ *Context*: IoT devices are usually scattered geographically, sometimes hard to reach and large in number. Operation managers and developers must be able to reconfigure devices or provision new ones in an efficient way and have pre-configured nodes.

♣ *Motivation Forces*: Suppose you have designed a system to display advertisements on some billboards spread in a region, each controlled by an IoT city hub[58]. At some point, you need to replace your technology stack entirely and provision a new environment remotely. You may also want to add new devices and provision their runtime environment and applications quickly.

It should be able to provision new devices in a way that can be repeated and produce the same results, so that it can be automated. This also contributes to keeping devices fault tolerant via a rollback to the known-good working state of the environment or applications quickly, if provisioning fails.

♣ *Solution Details*: Container-based virtualization is a good choice for provisioning resources, as they contain not only the code but also all other software dependencies, configurations and the whole runtime environment. By transferring the containerized image to a new machine and running it, we have a pre-configured environment with required applications installed along with any software dependencies.

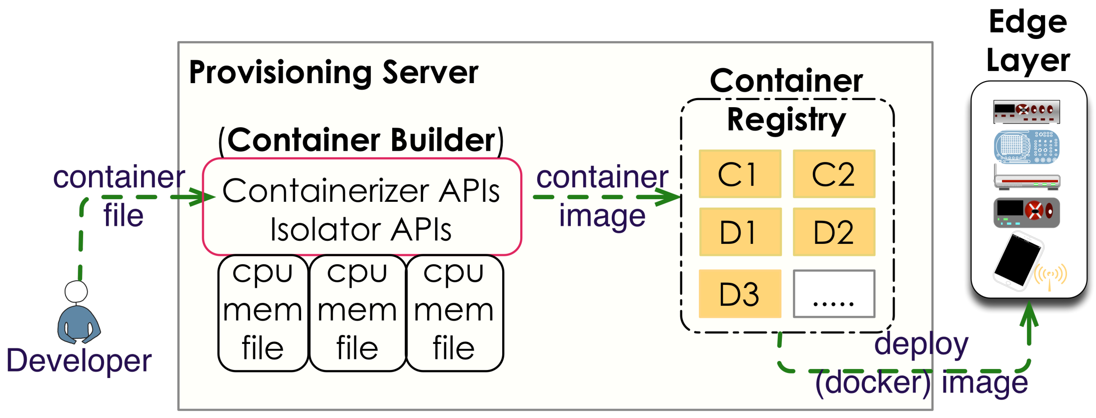


Figure 6.1: IoT provisioning pattern sketch.

For instance, provisioning and automated configuration can be done using tools like Puppet[1], Chef[2], or using Docker files, with which the Docker image is built and then transferred to the devices. Docker images utilize a layered and versioned file system, which has two benefits. First, the devices can only pull the layers they need and not the whole image; second, they can rollback to the latest or any working version of the image in case of failure or need. As images are static and read only, this leads to an incorruptible environment for the devices as a backup.

Docker images are built against a Docker file or other aforementioned automated configuration manager, which describes steps needed to build an image as commands to be run inside the container per line of the Docker file. Each step creates a new layer. The image is built; therefore, any resource-intensive step, such as compiling or downloading large files, is done once. Once the image is built, it is pushed to the central Docker registry/hub so that devices then pull the whole image or only the missing layers. These configuration files, like Chef recipes, Puppet manifests or Docker files can be kept as

[1]https://puppetlabs.com
[2]https://www.chef.io

documentation of the steps of provisioning under a version control system such as Git, so that a new commit can trigger the container builder system to build a new Docker image. The edge application together with its environment configuration can be kept in the same Git repository. Therefore, any update to the code or its environment will build a new Docker image and is transferred to the geographically distributed devices, using the same mechanism that is used for deployment. When the image is delivered to the end device, a new container is created using the image. A sketch of an IoT provisioning pattern is shown in Figure 6.1.

6.5 Edge Code Deployment Pattern

♠ *Problem*: How can developers deploy their code to many IoT devices automatically, quickly and safely, and configure them without being concerned about the long process of build, deployment, test and release?

♠ *Context*: Maintainability is a main factor while deploying a piece of code to some remote IoT devices. As developers enhance and improve the code or fix some critical bugs, they expect to deploy the updated code to their several remote IoT devices quickly. This grants distributing functionality between devices. Also, at some point developers need to re-configure the application's environment.

♠ *Motivation Forces*: Suppose you have designed a system to display advertisements on some billboards widespread in a region. You need to update the text or graphical features frequently or change the duration of ad display. Maintainability and adaptability are the most important challenges in such designs. You must be able to update the code and deploy it to all your devices at once.

Regarding the poor and flaky Internet connection (e.g., 3G) of a majority of IoT devices, it is best to only deliver the changes and not the application as a whole across the constrained network.

Also, developers should only be concerned with coding, as well as the tools they are familiar with. The tools for deploying the code to devices should be transparent to the developers. This leads to a fully automated deployment. Once automated, the safety of operations increases. The pipeline includes building the application, its deployment and testing and finally releasing and distributing it to edge devices. As for testing, the created image is a production-like environment and is used for testing. Once tests pass, edge devices can pull the image or the corresponding layer. The image is created against all the source code, the container specification and configuration files. Another point is that the developer should be able to rollback its deployment to an earlier version on-the-fly to avoid outage, which is crucial in case of IoT devices.

Moreover, the deployment process should consider and contain any software dependencies or configurations the new code needs. As such, the developer should be able to re-configure the application's environment or the overall technology stack remotely and safely to ensure consistency.

♠ *Solution Details*: As developers are familiar with version control systems, it is best to utilize it for deployments too. Nowadays, Git has become the de facto standard for developers to share their code and maintain versioning. It can be used as the starting point to trigger the build system and then the deployment process.

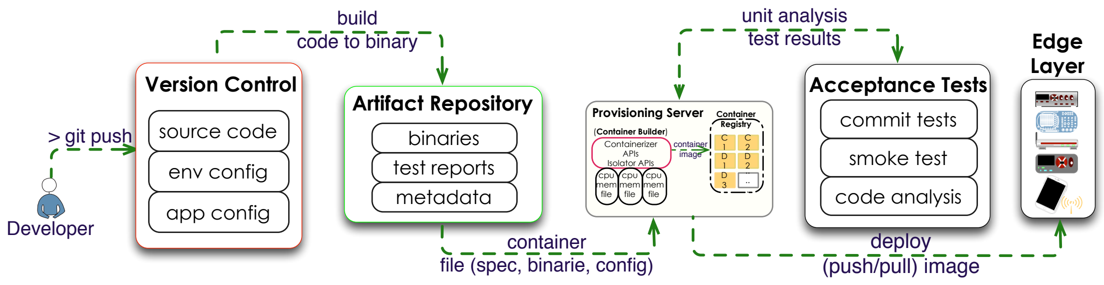


Figure 6.2: Edge code deployment pipeline pattern sketch.

Git can be utilized by developers to push a specific branch of code to a remote Git repository on the server and is notified of the new version of the software. Then using hooks, it can trigger the build system and start the next step of deploying the code to the devices. The build server builds a new Docker image and pushes the new layers of image to the central Docker registry/hub for the devices to pull. This way the developer only needs to use Git or any other version control system as the only familiar tool for deploying the code into geographically distributed devices.

Devices can periodically ask the central registry/hub for new versions or the server can notify devices about a release of a new image version. Then the devices will pull the new layers of image, create a container from it, and utilize the new code.

In summary, as illustrated in Figure 6.2, when the code is modified, it is committed and pushed using Git. Then a new Docker image is built and transferred to devices. Devices use the image to create a container and use the deployed code. The deployment pipeline is started with each commit, and changes in the source code are published to all edge devices.

6.6 Edge Orchestration Pattern

♦ *Problem*: How can we orchestrate IoT devices in accordance with their tightly scripted configurations as nodes of a cluster remotely? How can edge cluster nodes discover services?

♦ *Context*: Enabling a large number of devices connected via edge layer means empowering the cluster to manage its nodes to check their health state, their services state to reconfigure them. Moreover, in case of IoT devices to adapt and to calibrate nodes remotely and quickly. Furthermore, we want to be able to manage and run services in the cluster or schedule tasks on certain nodes and enable them to discover the services they need and re-configure themselves accordingly. Edge nodes in an IoT cluster should be able to find themselves and advertise services they provide to each other. Such configuration can be updated over-the-air via WiFi.

♦ *Motivation Forces*: Suppose you have designed a system to display advertisements on some billboards widespread in a region. You have devices to control each billboard. You want to be able to check their state and health status, manage them, check their services, change their runtime configuration, and execute services in the cluster or on certain devices.

The architecture should avoid having a single point of failure. Each node must be able to know the state of the whole cluster, as well as the state of each service provided by other nodes. In addition, it should manage and adapt itself so that horizontal scaling, replacing nodes and/or adding new nodes or services as ad portlets, in this case ad providers, remains simple.

Such changes in the cluster take effect once nodes are able to advertise their roles and services, as well as to discover each other. Furthermore, we need solutions to allow nodes to reconfigure themselves dynamically according to such changes.

Composite edge applications consist of several inter-related constituents microservices, each in need of its own environment, configuration and even a device. For instance, a home automation system is a composite application. We need a declarative way to describe the whole topology and deploy it considering the orchestration of the components, the services it provides and depends on, without configuring and installing each component separately on every device.

♦ *Solution Details*: Edge infrastructure toolkits treat and provision edge devices with limited compute resources (CPU, memory, and power) as constrained nodes of a cluster. Service discovery mechanisms can be leveraged by nodes to find each other, and the services they provide. Such discovery can also be achieved via device pairing. Once paired, devices trust each other and start sharing data or trigger functionality over a constrained network.

Containers' *compose-oriented*[3] technology enables us to deploy composite applications, since they can orchestrate multi-container IoT microservices. This mechanism expresses the composite application topology along with its specification in a declarative manner. The cluster manager can deploy and orchestrate the composite application for us, according to a service topology specification. This specification describes the composite application, its micro-services and the inter-relationships between them. The cluster manager, which hosts nodes in the cluster, receives the applications' specification and runs services on each node accordingly.

For configuration purposes, distributed `[key,value]` stores can be used. The distributed store, which does not rely on a single node for storing its data prevents a single point of failure. Besides, it enables the cluster to manage itself not only upon node

[3]https://docs.docker.com/compose

failures, but also upon cluster manager failure. Nodes advertise their services and their state by putting values into the store, others can retrieve the values/updates and establish pairing in one go. This may mean that the values change propagation time could be shortened in TTLs[4] if the nodes get notified of the changes and receive events, instead of polling for updates.

For tooling set, since nodes are resource-constrained, cluster managers like Apache Mesos[5] seem heavy, as they need a Java runtime environment to run. On the other hand, Fleet[6] seems a proper choice, as it can be run from a single binary. Fleet manages the entire cluster by controlling the `init` systems of the nodes. By communicating with the init system of the nodes, we can check if nodes or their services are up and running and run new services across the cluster or on specific nodes. Also we can define events based on devices or network changes and schedule tasks this way.

In a Fleet cluster, each node can be the manager, hence on its failure a new manager will be elected in the cluster. Nodes find each other and their services using service discovery mechanisms, the `[key,value]` store, or device pairing mentioned earlier. In this way, replacing nodes becomes seamless and scaling would be cheap. If Fleet is used, the composite application can be specified and described as service unit files. In the unit files one can describe the service, its physical place or the node one wishes to deploy the service on, its Docker image and the relationship between them. Then, Fleet will receive these service unit files and deploy the composite application on the devices in the cluster accordingly.

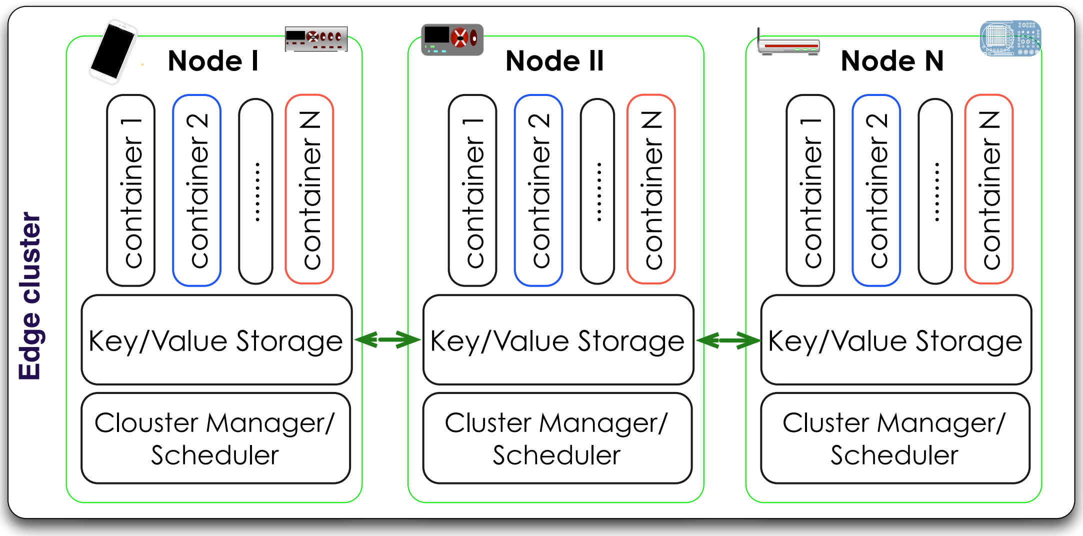


Figure 6.3: Edge cluster orchestrator pattern sketch.

[4]https://en.wikipedia.org/wiki/Time_to_live
[5]http://mesos.apache.org
[6]https://github.com/coreos/fleet

There are a number of choices for distributed `[key,value]` stores to be used as a service discovery mechanism; `etxd`[7] and `consul`[8] are preferred. Since they are quite light and written in Go, and compiled and linked statically, they can be run from a single binary without any dependencies. Contrary to other alternatives such as Zookeeper[9], which needs a Java runtime environment to run. Also they can be used along with cluster managers such as Fleet. Moreover, such systems expose a REST API and DNS service so that it is relatively easy to get or put values.

Next, `confd`[10], can be used if a conventional application uses environment variables or files for storing/maintaining its configuration. Confd watches for changes in a `[key,value]` store and updates configuration files and environment variables or even restarts services accordingly. So, nodes can reconfigure themselves on changes in the cluster.

From getting the tooling up, we can monitor all of our devices, check the health status of each device, monitor the running services, run new services on all or certain nodes and schedule tasks. When a new value is put into the distributed `[key,value]` store by any node in the cluster, it will be propagated and replicated to all nodes. Then nodes are notified of the new value and discover the services and their state. They then will re-configure themselves. So nodes can advertise the services they provide, find each other or change configurations upon discovery. Note the pattern sketch in Figure 6.3 depicting the built-in orchestration model in each node. A downside of this pattern is that the cluster should remain synchronized, meaning nodes should talk to each other periodically. Therefore, network chattering occurs a lot, which is of significance if networking is not economical.

6.7 Edge Footprint Messaging Pattern

◇ *Problem*: How can edge devices in an IoT system send and receive data through an efficient and real time data bus?

◇ *Context*: Many IoT devices need to send and receive data, while they are usually connected to low quality or expensive networks. Therefore, protocols and patterns to ensure reliability, quality, security and efficiency are essential.

◇ *Motivation Forces*: Suppose you have a system of advertisement billboard widespread in a region each on controlled by an IoT device. You want to be able to send a message from the devices each time the next ad starts to have a heartbeat of the system and know its status on a central dashboard. In some other cases, these devices stop displaying ads when they receive the shutdown message. You want to be able to send the shutdown message to all or a group of devices.

[7]https://github.com/coreos/etcd
[8]https://www.consul.io
[9]https://zookeeper.apache.org
[10]https://github.com/kelseyhightower/confd

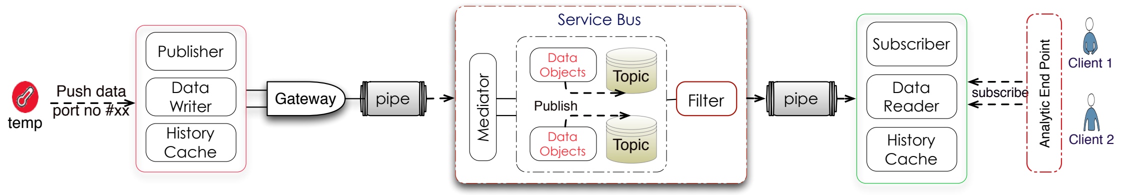


Figure 6.4: Edge footprint messaging pattern sketch.

We need a common protocol or format for exchanging data between devices so that we can use a middle-ware as our edge service bus to dispatch and distribute messages between the edge devices according to their need, functionality and purpose. As IoT devices have constrained resources, exchanged message format must be efficient to process, transfer, digest and produce. So the message format, its size and the time complexity to process it must be considered. Security of these formats must be considered too. The data emitted by some edge devices, such as sensors, must be kept private, and only some of the devices can have the authority to control sensitive messages.

Quality of the messages and their integrity must be ensured to prevent retrieving and analyzing false data or sending wrong instructions to devices, which may result in irrecoverable consequences. Messages should be delivered in real time with minimum delay from and to devices. This pattern advocates a sustainable and scalable IoT application life-cycle.

◇ *Solution Details*: An edge service bus can be utilized and implemented to be connected to all edge devices with an appropriate protocol for each device according to its place, objective and network connection quality.

Publish-Subscribe is an appropriate choice when there are many edge devices waiting to react when a certain event occurs or message received. Each subscriber will listen to certain desired topics. In case of our billboards, each geographical region is a topic and devices in each region are subscribed to their corresponding region topic. So we can send a shutdown message to a topic to shut down billboards in one region.

In a real time Publish-Subscribe pattern, components like publishers, subscribers, topics, service bus, endpoints, data writers and data readers are required. First each publisher can send data through data writer. Each publisher has at least one or many data writers to send or write to a component, called topic. Data writer is an interface which is used to write data objects into topics. There are two types of endpoints: (i) one endpoint acts as a sender or publisher and (ii) the other one acts as a receiver or subscriber. The first type of endpoint is the combination of a publisher and a data writer. Each data writer accesses only a single topic. The edge service bus can use mediators to facilitate exchanging data between devices with different protocols or to modify the exchanged data. Mediators are also capable of filtering, decryption and transformation.

Filtering messages on the writer side (in the service bus) has some advantages. For instance, a data object that would not pass any reader will not be sent to save bandwidth. Also a writer can include information on what filters it passed. After filtering and transforming data objects, they will be moved to a data cache. The Data cache is used to prevent data loss. Filtering can be done on the reader side too, specifically when writers do not apply any filtering. The receiver endpoints subscribe to their desired topic. As soon as encrypted data objects are stored in the data cache they will be sent to the subscribers.

There are various protocols which can be used for IoT devices to exchange data, such as MQTT[11] and CoAP[12]. These protocols consider constrained resources devices and limited network connection. Therefore they are preferred over other protocols such as HTTP. MQTT is more a messaging protocol for pub-sub pattern with topics, while CoAP is more an HTTP-like protocol with content type and RESTful methods.

Each of the two popular protocols has its own capabilities and advantages. CoAP provides an interoperable solution for communication between tiny resource-constrained devices (sensors, actuators, controllers, etc.) which communicate over resource-constrained networks. Reducing the message header size and response codes, minimizing the set of methods for exchanging data, providing both confirmable and unconfirmable messages, and the possibility of dividing large messages by using the block transfer algorithm, make CoAP a lightweight and practical protocol for communicating resource-constrained devices.

In addition to request/response capability, CoAP has an extra extension for providing subscribe/notify pattern (event subscription pattern). In subscribe/notify pattern, subscribers, subscribe to an event from a special endpoint (resource), on the other hand, the endpoint notify subscribers in the case of occurring the interested event.

In our case we can apply CoAP to each device which communicates through UDP. When the next ad starts, a notification is published for all subscriber devices. Subscribers start or stop their jobs by receiving the corresponding notification.

MQTT is based on publish/subscribe pattern and is done over TCP. In this model there are three roles: publisher, subscriber and message broker. Publisher connects to the broker and publishes its message (topic), on the other hand, subscriber connects to the broker and subscribes for the interested topic. The role of broker is to relay published topic between subscriber clients. As communication in MQTT is done over TCP and messages are acknowledged, the messages are delivered once and in the order, so it can be more reliable for more sensitive cases.

There are a number of formats for serializing and exchanging data such as XML or JSON, but CoAP and MQTT are preferred since they are cheaper due to the expensive networking in IoT devices. CBOR[13] is a data format for data serialization similar to

[11]http://mqtt.org
[12]http://coap.technology
[13]http://cbor.io

JSON and is more efficient in terms of size. It also supports binary data type which is usually used by sensor data on IoT edge devices.

Clients read and write data which the transformer components adapt and translate it in mediator layer. This solution solves the problem of programming language and hardware dependency of IoT edges. This capability hides connectivity and topology details from clients and loosens coupling. The edge footprint messaging pattern sketch is summarized in Figure 6.4.

6.8 Edge In-Memory Data Retrieval Pattern

♥ *Problem*: How can we store and retrieve large amounts of data emitted by edge devices?

♥ *Context*: IoT devices are not able to store total data since their resources are limited. Therefore data should be stored in storages with more capacity. Nevertheless IoT devices are generating data in every moment so the data retrieval method should be able to work in two speeds: First the speed of generating IoT devices and second in the writing speed of its storage.

On one side, our resources are limited. On the other side, we face big data. IoT devices generate an enormous corpus of data in an ordered interval while their memory is limited. As a consequence, they should free their memory from old data immediately to be able to receive new data. Hence we face limited memory and huge amount of data.

♥ *Motivation Forces*: IoT devices get data via their sensors from the environment and put them into their memories. These memories have a limited storage space while being fast. So data should be sent to another storage.

This storage should be a cache system, which gets data from IoT devices, and puts them into a database. Cache systems consist of fast memories. They are able to receive data as fast as possible. But these memories are limited, so we should scale them up, to be able to gather huge amounts of data. Replication or other cloud clustering solutions can be incorporated to eliminate single point of failure and enhance safety, security and speed.

♥ *Solution Details*: Distributed cache is a pattern of storing data on multiple servers in order to provide quick data retrieval. Distributed cache systems are usually used on web servers and application servers to support non-local storage (remote storage) as a fast and extendable data storage.

We will use a distributed cache system for storing IoT generated data, though it might be costly, data consistency and high speed is assured. Our pattern will improve response time and solve the problem of storage capacity of IoT devices by reducing data access latency. IoT devices put data not in a single cache but in distributed caches. Putting data in a distributed cache system decreases the possibility of losing data. So every node of the distributed cache system has its own data and some backups of other nodes. Via this method if one of the caches gets lost, another one would be able to load its data to be placed in the writing queue or to be read by clients.

Indexing decreases query execution time and enhances performance of the overall system. In this context, to increase performance, this pattern would contain some additional data in NoSQL database for indexing. There would be no indexing in caches since they are only fast memories that retrieve data from a persistent database and make it available to users. Via indexing, query time decreases while data retrieval performance enhances. Distributed caches can be configured in different ways. We could configure them to support replication to increase the availability of replicas, or we could set different algorithms for persistence of data on NoSQL. There are some different strategies for management of replications. The structure that we will suggest here is a decentralized model, where all nodes store a replica of another (primary and secondary) model and they will update each other. The other method that could be used is a strategy that all nodes store the replica of all nodes except itself or we could use a strategy where a centralized node stores the replica of all nodes.

Also a strategy for creation and placement of replicas must be considered. We suggest the dynamic replication method for this purpose. It is a hierarchical model, in which the data management system keeps track of all available storage on all server nodes in the cluster and also stores the users that request a data node. When the number of hits for a specific data in a node increases, the management system will replicate the data on the server that directly services the user. As detailed below, we present three effective distributed cache systems algorithms. Meanwhile, it is an indication that the way in which an algorithm is implemented can also have a significant effect.

6.8.1 Cache Aside

In this model, as illustrated in Figure 6.5, the client is responsible for both read/write to cache and database. Database and cache do not interact with each other.

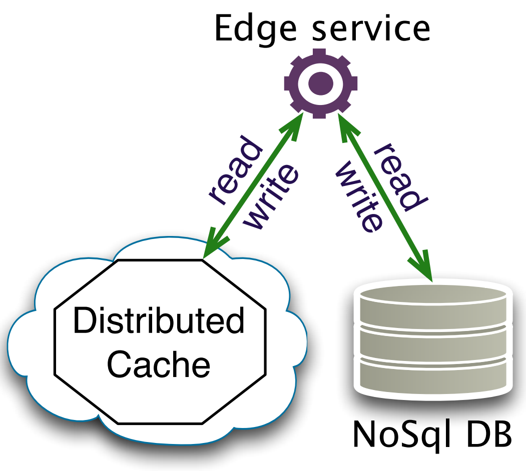


Figure 6.5: Edge cache aside model.

6.8.2 Read Through/Write Through

Here, client would read/write data to cache and the cache is responsible for synchronizing data with database. This model is shown in Figure 6.6.

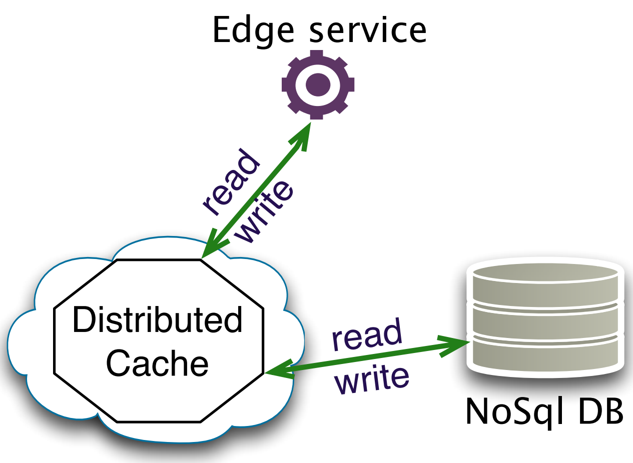


Figure 6.6: Edge cache read/write through model.

6.8.3 Read/write asynchronously

In this method, as depicted in Figure 6.7, read and write processes will be done with a predefined delay and in an asynchronous manner.

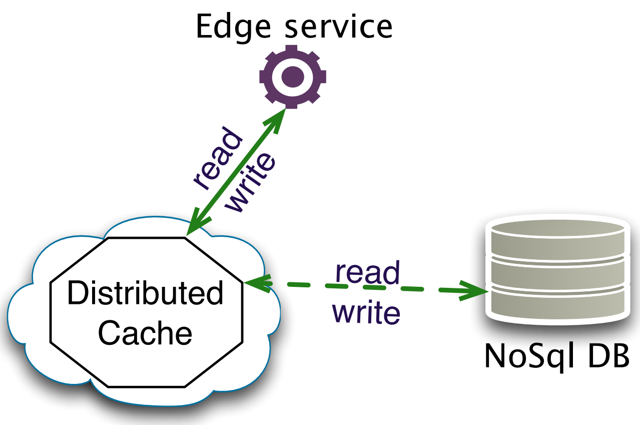


Figure 6.7: Edge cache read/write asynchronous model.

The last method is the best for this pattern, since edge devices can read/write the cache and the cache will persist data on a NoSQL database. For speeding up this asynchronous transmission of data, we use a queue to let the cache system write the data at its own speed and let the NoSQL database fetch it at its own speed. Also, we should mention that loading data from a NoSQL database is not possible. In order to load data from the database, we may load it to the cache making it available for clients to read data from cache. Here are the steps and details of a query execution:

- **Query cluster:** The client must run a query service in order to be able to get the query from the cluster.
- **Matching objects:** The query service gets matching object count from all partitions. Count here is finding the number of matching objects in the query via the in-memory indexes
- **Load data:** The query service must decide to load data from the DB if it is needed. The data does not exist in partitions and is persistent. As was mentioned, the DB could be optional.
- **Merge result:** The query service will get the result from all partitions and it will merge the results received from each partition separately.

Via this pattern, we can store edge data in an appropriate data storage (NoSQL database). The Cache system makes reading faster than other patterns, but writing is not as fast as reading. After the data is written to a cache, it will be persisted on a database. So, there is an enhancement in writing performance. In the end, this pattern is fast-enough in writing and reading, but data should be stored in an outer storage, which is not a cache memory.

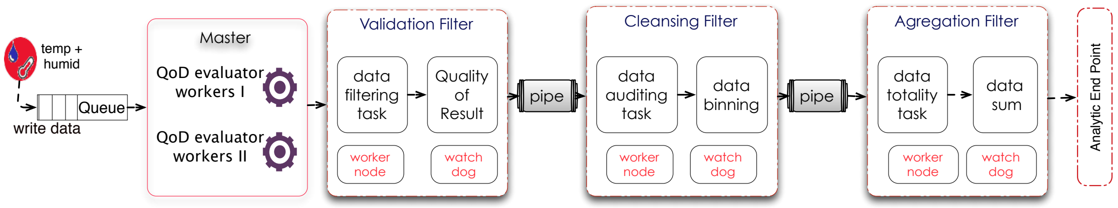


Figure 6.8: IoT In-Device data preprocessing pattern sketch.

6.9 In-Device Data Preprocessing Pattern

♡ *Problem*: How can data scientists ensure that the raw emitted data from edge devices meet the baseline quality of data (QoD)[14] for further processing?

♡ *Context*: The edge data stream has a long data-processing workflow in terms of collection, storage, and processing. In effect, the decisions made at the earlier stages of

[14]According to TechTarget, data quality is a perception or an assessment of data's fitness to serve its purpose in a given context.

the flow can significantly impact the processing at later stages. This raises interest in processing at the edges to extract actionable insights from data. Data captured from the edge devices such as temperature/humidity sensors tend to lack semantics and be very noisy. In effect, it is optimal to perform in-device lightweight data preprocessing. This saves time to create meaningful outcomes from data.

♡ *Motivation Forces*: Suppose we have a system of advertisement billboard spread in a region. The advertisement is designed to change dynamically with respect to the city weather forecast. For instance, the hub is equipped with an embedded humidity/temperature sensor that captures weather dynamics data and transmits it to the advertisement microservice. The running advertisement service retrieves sensor data, correlated with geographical location, demographics, and weather information. It gives the city hub collective intelligence to act eventually by adapting its message accordingly. In effect, if some raining weeks are predicted and confirmed from sensor data, then the billboard will advertise the appropriate products like umbrellas.

To achieve this, the city hub will employ a lightweight data processing on the sensor data via the hub's on-board computing resources. This contributes to proper dynamic decision making proactively.

♡ *Solution Details*: The IoT edge devices are considered resource constrained environments that affect the processing of tasks and might affect the performance of running tasks if resource elasticity constraints are not met. In this pattern, we are offering an elastic task flow system for an IoT sensor data-stream analytics process, which allows for building simple pipelines of data processing tasks. The task flow system will proceed with real-time processing of a sensor's data-stream feeding the dependent services. Raw sensed data may lack accuracy and completeness due to device malfunctioning. This poses a challenge in processing data and affects the correctness of analytic results.

An IoT big data analytic process is a pipeline, which is composed of a number of tasks. The tasks used for ensuring quality of data are the early stages of this pipeline ranging from simple cleansing to binning and restoring of corrupted data points. The later stages include data intensive analytic processes which need more computing resources such as CPU, memory and storage and should be done in a data center. But the early stages consist of lightweight tasks that could be done on devices.

The core idea of this pattern, as illustrated in Figure 6.8, is based on *Pipe* and *Filter* architecture. There are three filters in lightweight data processing: data points *validation*, *cleansing*, and *aggregation*. These simple operations are wired together as a Directed Acyclic Graph (DAG) of tasks. The city hub's IoT platform handles tasks dependencies, task flow management, state management, scalability and policy-based resource allocation to the running steps. Each task might be a Hadoop job in Java, a Spark job in Python, or a simple text-processing Python snippet. To achieve this, we need to employ the *Edge Provisioning* pattern to be able to deploy the pipelined tasks on the sensor-equipped IoT edge devices.

In Pipe and Filter architecture, the output of one filter is fed, as input, into the next filter. In this pipeline the first filter does data validation. The task is done by submitting a validation job to a spark master. The validation job is done by spark workers. The next filter is cleansing. Based on the different cleaning policies, missing and invalid values may be deleted or replaced with a reasonable value. Data aggregation is the next. The output of these phases must satisfy the quality of data constraints. The built-in monitoring daemon will observe the elasticity weight of the running tasks in terms of underlying resource usage. This contributes to an automated resource leveling of each task of the deployed data-centric pipeline. Moreover, you are able to cope with prospective aspects of elasticity like resource allocation over future demands and supplies. At any given time t, Edge platform provides enriched pieces of information

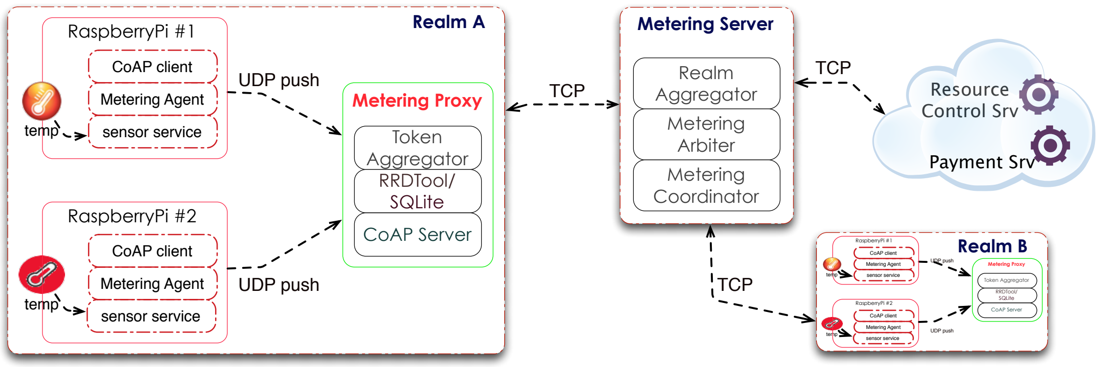


Figure 6.9: IoT Diameter of Things (DoT) metering pattern sketch.

about the IoT applications temperature/humidity observations. Such observations can be represented as a column-vector $\mathbf{o}_t \equiv [v_{t,1}\ v_{t,2}\ \dots\ v_{t,n}]^T \in R^n$ of metrics data stream values at time t. The stream of sensed data can be regarded as a frequently expanding $t \times n$ matrix $\mathbf{O}_t := [\mathbf{o}_1\ \mathbf{o}_2\ \dots\ \mathbf{o}_t]^T \in R^{t \times n}$ where the new incoming streams are added as matrix rows at each time interval t in real-time. In our IoT case, $\mathbf{O}_t$ is the measurements column-vector at t over all the metrics, where n is the length of the vector and indicates the number of metrics and t is the measurement time-stamp. These vectors represent the set of measurements obtained for the n metrics at a specific observation. In particular the rows of the matrix represent various `monitoring observations` in a given period, while the columns are the sample `values` detected for each metric during the observations.

$$\mathbf{O}_{t,n} = \begin{pmatrix} edgeID_{1,1} & appID_{1,2} & \cdots & temp_{1,n} \\ edgeID_{2,1} & appID_{2,2} & \cdots & temp_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ edgeID_{t,1} & appID_{t,2} & \cdots & temp_{t,n} \end{pmatrix} \tag{6.1}$$

These vectors represent the set of measurements obtained by the n sensor at a specific observation. In particular the rows of the matrix represent the different observations in a given period, while the columns are the sample values detected from each sensor during the observations. This equation 7.1, formulates a simple formal representation of sensor data.

6.10 Edge Diameter of Things (DoT) Pattern

♧ *Problem*: How can IoT service provider monitor and meter the actual usage of IoT deployment units in real-time or near-real-time, in order to monetize them? How the IoT composite application resource usage, as well as the service usage can be charged against a specific user balance?

♧ *Context*: From the provider's perspective metering mechanisms can vary based on applied business models. These mechanisms range from different usage patterns such as invocation basis (event-based) and usage over time (time-based), to subscription models such as prepaid and pay-per-use models. This yields to the need for defining some metrics for service and resource usage, which in turn, can be used to measure the consumption of the service and to price it.

♧ *Motivation Forces*: Suppose you have provided an IoT platform, which presumably consists of the hardware (device) and composite IoT services, to your customer, and you would like to monetize it based on the real usage of the client, while guaranteeing a fare transaction for both ends. To achieve this goal you need to measure the rate of actual resource and service utilization, as near real-time as possible.

You may provide your service in a prepaid model, where you and the client agree on certain amount of credit to be reserved prior to service delivery. In this subscription model, you should ensure that the actual usage does not exceed the reserved credit on one hand, and on the other hand, the service delivery does not terminate while there are still some credits remaining.

You may provide the edge service in a pay-per-use model, where you charge the client based on actual usage of the IoT platform. The actual payment and therefore the subtraction from user credit should be done at a certain rate. This model can be bounded to a definitive time limit or it can be unbounded.

Your provided service is normally a composition of several microservices of different kinds, and therefore various charging schemes are applicable to them. For instance, some services are event-based, where the charging scheme is based on the number of service invocations. Other services are time-based, where the duration of time the service has been in use should be considered. Furthermore, for time-based services, you might want to monitor underlying resource usage of the service as well. The resource usage monitoring can again be time-based or it can be divided to more granular scheme of monitoring memory, CPU, filesystem and bandwidth usage of the service.

Furthermore, the IoT platform normally resides on the client side and your access to the edge device is limited to an Internet connection with a certain bandwidth. In this regard, you should prevent overwhelming the network by transferring too many monitoring messages back and forth, while still supporting the near-real time monitoring of the usage.

♧ *Solution Details*: A light-weight metering protocol can be utilized to support the telemetry of composite IoT applications deployed on resource constrained devices.

The prerequisite to such a solution is to define a specific agreement called "metering plan", offered by the provider and accepted by the client. The subscribed metering plan is an indication of all assumptions that need to be considered for a proper service delivery, continuous and near-real-time usage metering, and the subsequent charging. The plan includes: i) subscription type (i.e. pay-per-use model or prepaid model), ii) list of constituent microservices provided to the client, iii) usage pattern and measurement unit of each service. A time unit is used for duration-based services and number of invocations is used for the services with an event-based usage pattern. iv) the price for each allocated service unit, v) the price for underlying resource usage, vi) the resource Used Unit Update (U3) rate for each service. vii) if prepaid subscription type is selected, then the maximum allocated units to each service is also included in the plan. viii) subscription-fee for prepaid model, and subscription time for a bounded pay-per-use model.

The U3 rate defined in the plan, is the rate of producing usage tokens for each microservice. The actual U3 rate for a prepaid economic model will be calculated based on the maximum allocated units to each constituent service. e.g. The U3 rate of 20% for a duration-based service with maximum granted units of 100 hours will cause to send usage updates every 20 hours. The same U3 rate for an event-based service with granted units of 100 invocations will result in generating a usage update after every 20 invocations. The U3 rate for a pay-per-use model, on the other hand, is calculated merely in time units, regardless of service type. This means, for a defined U3 rate of 5% for a service bounded to 1 month of usage (as defined via subscription time in plan), the actual usage report will be sent every 1.5 days. For unbound pay-per-use model, U3 is simply the rate of sending updates of each service per time unit, e.g. every hour.

The U3 rate is one of the main factors defining the real-time characteristics of monitoring underlying services. Nevertheless, it exerts an impact on network congestion rate. Consequently, it is crucial for the service provider to assign an appropriate value to it to balance the trade-off between generated network traffic and real time update.

To realize the metering infrastructure, two main components are introduced: (i) the metering server, which is a central component, is responsible for telemetry coordination. (ii) the metering agent, which is a distributed component residing on the edge device and assigned to a microservice, is responsible for collecting usage information and sending it to the metering server. As soon as an IoT application is instantiated for a specific user, the metering server receives a copy of the metering plan. It then parses the plan and calculates the U3 rate for each constituent service in the plan. Together with the calculated U3 value, it then sends the request to start metering each service to its newly assigned metering agent. At each U3 interval, each agent will send the actual service usage, as well as the resource usage to the metering server.

In order to lower the load imposed on the network by sending usage token messages from the metering agent to the metering server, two supporting solutions are leveraged. First is the use of an intermediary component on the edge side, called the metering proxy, to aggregate several usage tokens within a window frame of predefined time or message count. Second is the use of a low overhead messaging format to transfer the data from agents to the metering aggregator. To this end, CoAP is utilized, which is specifically designed to work in constrained networks of IoT environments.

As the usage update summary is received by the metering server, it is used to calculate the charge accordingly. Figure 6.9 provides a schematic view on architecting DoT's metering collaborating components. Our DoT protocol[59][15] is currently in development and the ongoing draft is available in IETF as an Internet-Draft submission.

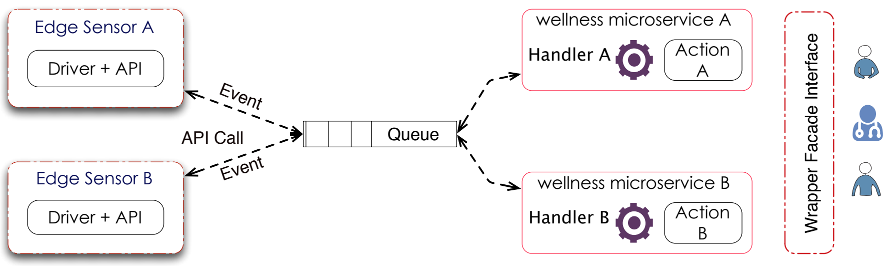


Figure 6.10: IoT Wearable Facade Pattern sketch.

6.11 Edge Wearable Façade Pattern

♤ *Problem*: How can a wearable device, sensor, or a micro robot interact with humans while the device level knowledge is not required? How can the low-level APIs be encapsulated from the human sensible perspective while (s)he needs to communicate easily with the wearable device? How can the meaning of a micro intelligent widget be embedded in micro wearable sensors?

♤ *Context*: Edge devices and sensors usually expose APIs in the form of libraries. These libraries provide the sensor or device functionalities which can be integrated into more coarse-grained modules. Working with such sensors and making them integrated into a sensible device requires the knowledge of detailed data-sheets and API documentation. This contributes to higher level functionality.

♤ *Motivation Forces*: In patient health-care systems, there is a need for some wearable devices to gather vital patient information to find out the patient's health condition for proper caring support. These wearable devices such as smart T-shirts, smart belts, smart

[15]https://datatracker.ietf.org/doc/draft-tuwien-dsg-diameterofthings

watches and smart glasses produce vital signals and communicate with electronic health record systems. Each wearable device is equipped with a network of sensors, which each providing a single decoupled function. In such cases, devices implement a human-based sensible interface so that individuals can utilize the high level functionalities efficiently.

♤ *Solution Details*: Edge Wearable Façade pattern exposes unified interface to complex sensor libraries and APIs. This interface can interact with the sensor layer APIs and make them easier to understand and abstracts the complexity and dependencies away from the user view. Therefore the complexity of sensor APIs, sensor communications, events, etc. is transparent to the user view.

This pattern uses an event driven architecture. In the lowest level, each device exposes some functionality via the built-in APIs. Each device has implemented its logic via a microservice so that the device functionalities can be invoked. There are two streams of data, one is the commands from microservices to device/sensor and the other one is the device/sensor signal, which we see as an *event* in this pattern. The microservice sends commands and the device/sensor sends events. In the microservice layer, there is a predefined handler for each event. When an event gets triggered by a handler, an action will be invoked. We offer the use of lightweight queue-based messaging for in-device communications.

In summary, as shown in Figure 6.10, the Façade Interface Wrapper which is on the top of the architecture hides the complexity of the device layer and their communications and APIs in to a meaningful high level interface. A simple use case of this interface is an EHR (Electronic Health Record) system as a wellness function. The wellness function may use different micro-services and the corresponding sensors to evaluate the patient vital signs and do indispensable action.

6.12 Related Work

In relation to our approach, there are some similar, commendable work. For instance, the Pattern Languages of Programs (PLoP)[16] community has developed various design patterns, like pattern-oriented software architecture: a system of patterns[60] defining a pool of proven solutions on how to describe large-scale applications. The design patterns: elements of reusable object-oriented software[61] offers timeless and elegant solutions to recurring classes of challenges in software design. They also proposed proven techniques to achieve patterns for fault tolerant software[62]. Looking forward, the Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions is a profound book on how to think and design distributed systems asynchronously. Eloranta and Leppaenen[63] present three patterns that supports varying the control system tailored to fit the customer's needs. The contemporary shift to cloud, the authors shared their foresight in the Cloud Computing Patterns: Fundamentals to Design, Build,

[16]https://en.wikipedia.org/wiki/Pattern_Languages_of_Programs

and Manage Cloud Applications[64], in an abstract format that is independent of concrete vendor products.

Most of the current proposed patterns are focused from a cloud perspective. Few are designed with an edge-based focus in mind. Hashizume, Yoshioka, and Fernandez[65][66] build a catalog of misuse patterns like Resource Usage Monitoring and Malicious VM Creation. The Amazon Web Services (AWS) cloud patterns[67] documents design principles for applications hosted in the Amazon cloud. Microsoft proposed a set of Azure cloud design patterns[68] claiming to be usable for native cloud applications. The cloud architecture patterns[69] introduces 11 architectural patterns utilizing cloud-platform services. Erl et al.[70][71] present cloud design patterns very briefly. The only article written by Michael Koster[72], entitled "Design Patterns for an Internet of Things" , just defines some basic IoT concepts in a couple of sentences. No pattern format is supported. In contrast to all the noted related work, we took further steps towards identifying reusable edge applications design constructs.

6.13 Conclusion

In this chapter, the authors offered eight design patterns applicable for designing, building and managing IoT applications. IoT exposes underlying devices as software-defined things or edge applications. Engineering edge applications requires abstracted design principles that prescribe how collaborating things should work. From the engineering perspective, the proposed patterns cover, realizing all stages of the edge applications life cycle from their dynamic composition to their deployment and fulfillment.

So far, we have defined eight design patterns enabling IoT architects to construct edge applications. As an outlook, our future work includes further extension to the design patterns to support more diverse applications, as well as refining and updating existing ones. We will focus more on patterns to be used for elasticity, resiliency and Software Defined Networking (SDN) patterns for edge computing. This will study the behavior of impacting players, such as competing applications, in making decisions on the allocation of limited resources amongst infrastructure forces of supply and demand.

Just as evolving IoT offerings target a large diversity of systems, we envision that such design patterns may leverage the performance and scalability of edge applications as well as to gaining acceptance as a de facto design standard to give adequate foresight to edge engineers in IoT.

Part III

Micro Telemetry

CHAPTER 7

Telemetry of Elastic Data

7.1 Introduction

Utility computing[73, 74] is an evolving facet of cloud computing that aims to leverage and treat computing resources as a metered service, like natural gas. It enables a *Pay-per-use* or *utility-based* pricing model through *metered data* to achieve more financial transparency. Metering measures rates of resource utilization via metrics, such as data storage or memory usage, consumed by the cloud service subscribers. Metrics are statistical units that indicate how consumption is measured and priced. Furthermore, metering is the process of measuring and recording the usage of an entire application, individual parts of an application, or specific services, tasks and resources. From the provider view, the metering mechanisms for service usage differ widely, due to their offerings that are influenced by their cloud business models. Such mechanisms range from usage over time, volume-basis to subscription models. Thus, providers are encouraged to offer reasonable pricing models[75] to monetize the corresponding metering model.

In the cloud market, providers are expected to have a layered metering model together with its associated pricing schema to exploit various resource usage granularity. In effect, the more fine-grained the metering service is, the more transparent is the utilization. Data aggregation is needed to provide a broader view of resource usage and conversely, small-footprint or micro metering is required to achieve a more granular view of the resource utilization. Thus, increasing the resolution and precision of metering the underlying resource unit to improve the validity of the quantified cost appears to be vital. However, this comes with an elevated cost for monitoring, which we assume is reasonable. In this context, the underlying resource usage events are time-series data that are streamed at tiny time intervals (e.g., 3 seconds) to enable current and consistent data retrieval of a metered unit.

The quest for telemetry of the client's job resource usage becomes more challenging when the job is deployed and processed in a distributed model. For instance, the MapReduce framework[76] offers an abstraction that simplifies the execution of data processing. Such computation intensive applications run in a distributed setting while hiding the details of parallelization, data distribution, load balancing and fault tolerance. It aims to parallelize the data-intensive application processing and less focus is put on efficient underlying resource utilization. Enriching MapReduce with telemetry services will contribute to more optimized resource utilization and performance in the cluster. This leads to a question, how can granular metering contribute to more utilization value? The reason is granular elasticity control. Metering *Map* or *Reduce* tasks enables granular elasticity control on multiple levels by varying elasticity requirements like cost and quality[77]. The next question is how does granular metering improve performance? The solution is process-weight classification of the application jobs. With the classification framework, the job types are classified in terms of resource usage (map or reduce-intensive), and this seeds algorithms for an optimal tailored scheduling. This leads to a more optimized resource allocation than other current plugged policies. From the consumer view, clients

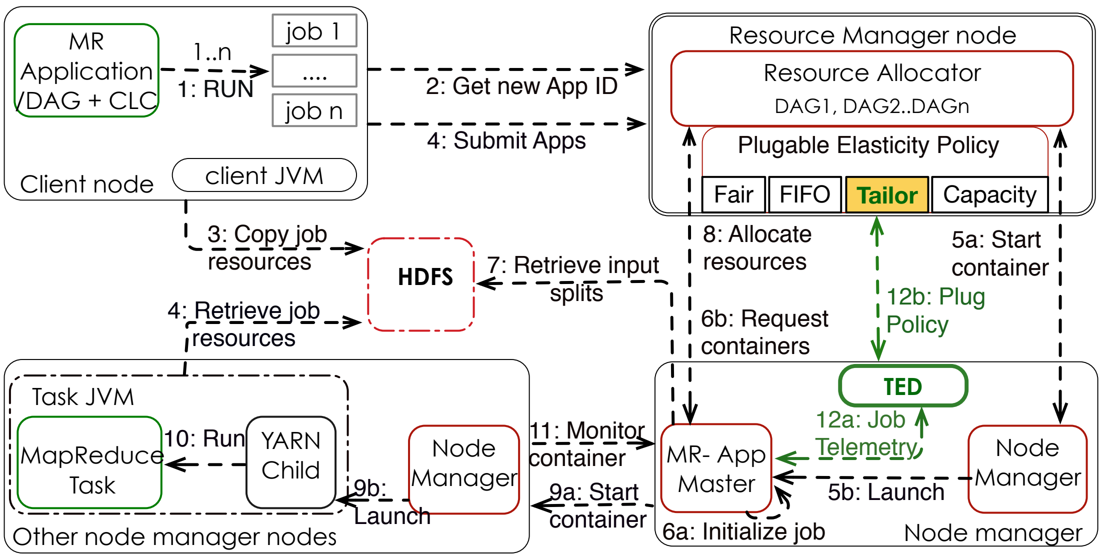


Figure 7.1: TED middleware data and process flow with their sequence of events in YARN

seek to economize their usage patterns by optimizing their map or reduce-intensive jobs. Along with these motivations, MapReduce-based utility computing contributes to planning a cluster's future resource requirements for more elasticity and performance. The flip side of the coin is that providers will gain an understanding of how their underlying resources are being consumed to bill users respectively. Thus, it makes eminent sense to meter emitted data of MapReduce resource usage.

Cloud Market-Leader Amazon offers an *Elastic MapReduce (EMR)* web service, for instance, a hosted platform on the Amazon cloud where users can instantly provision Hadoop clusters to perform their data-intensive tasks. Amazon EMR uses Hadoop, an open source framework, to distribute your data and processing across a resizable cluster of Amazon EC2 instances. Based on the Amazon EMR pricing[1] model, you pay an hourly rate for every instance-hour you use. The hourly rate depends on the instance type used (e.g. standard, high cpu, high memory, high storage, etc). Hourly prices range from \$0.011/hour to \$0.27/hour (\$94/year to \$2367/year). The Amazon EMR price is in addition to the Amazon EC2 price (the price for the underlying servers). There are a variety of Amazon EC2 pricing options, including *On-demand*, *Reserved*, and *Spot* instances. Besides, Amazon EMR uses other services such as Amazon S3, SQS, SimpleDB for its operations which are billed separately.

Due to the Amazon EMR pricing model, a 10-node cluster running for 10 hours costs the same as a 100-node cluster running for 1 hour. Kambatla et al [13] drilled this down for more transparency and showed us that resource elasticity for MapReduce jobs is not entirely symmetric, *i.e.,* 1 hour on 100 nodes may not accomplish the same resource usage throughput as 100 hours on 1 node. This poses a challenging decision of choosing the right cost-efficient cluster size. Following the previous scenario, there is a potential situation where the duration of job's execution time comes into play. If the job doesn't go into overtime, but runs for 61 minutes then you would get charged for the next hour, doubling the cost of the job! The most remarkable feature of EMR billing is that Amazon bills by an hourly increment. Cluster initialization seems to take 5 minutes or so with the Amazon distribution, so if one of your job's input paths is missing, it will take about 5 minutes to fail, but you'll be charged for the whole hour. You can definitely buy more machines and get a job done in less than an hour, but the cost-efficiency goes down. For instance, if your job completes in 30 minutes you will roughly double the cost.

However, in this context, the visible challenge for MapReduce jobs that must scale to extremely high capacity, is to ensure actual application resource consumption in terms of quality and cost elasticity. To address the above challenges, we implement an elastic data telemetry middleware called TED for YARN framework. TED collects "actual telemetry" events of running map/reduce tasks and then "meters" such data in meaningful way by returning the outcomes of its granular metering for two purposes. First, to generate billing statements to charge clients respectively and, second to generate and enforce tailored policies to control current applications' resource consumption behavior.

A mapreduce application can be represented by a directed acyclic graph (DAG). The DAG is used to represent a set of tasks where the input, output, or execution of one or more tasks is dependent on one or more other jobs. The tasks are nodes in the graph, and the edges identify the dependencies. The elasticity strategies can be applied on the whole DAG or parts of it. Similarly a set of DAGs can be bundled into a data analytic workflow driven by the Apache Oozie engine[2]. Some strategies might constitute constraints over the number of pending or running DAG jobs, maps or reduces.

As illustrated in figure 7.1, we have positioned and deployed our TED middleware in the

[1] http://aws.amazon.com/elasticmapreduce/pricing/

[2] http://oozie.apache.org

resource manager node on the YARN architecture. The figure also depicts the event flow sequence corresponding to the mapreduce job life-cycle together with its metering process. YARN dynamically allocates resources for MR Jobs as they run. Let us demonstrate how TED utilizes resource provisioning. In this scenario, at stages 1,2,3,4 as shown in Figure 7.1, a client submits a job or a DAG of jobs with the required container launch context (CLC) information to the resource manager and copy data source to the HDFS. As soon as the job is submitted, the *ResourceAllocator* negotiates a container and instantiates the *ApplicationMaster* for the job at stages 5a and 5b. At this moment, stages 6a and 6b, the *ApplicationMaster* initializes the job and requests containers from the *ResourceManager*. Then, at 9a, 9b, 10 and 11, it launches the granted container, runs the job and monitors its execution. The containers are configured based on the CLC specification. Our focus lies on stages 12a and 12b, where TED retrieves and meters the resource usage data, interprets the job's behavior and then plugs the suitable and tailored elasticity policy (scale-up/down) into the resource allocator for enforcement.

ApplicationMaster monitors the job execution flow through NodeManager, and observes their status and resource usage metrics (cpu, memory, disk, network). Then it negotiates resources for a single application (a single job or a directed acyclic graph of jobs). The ApplicationMaster initial request is structured in the `[resource-name, priority, resource-requirement, number-of-containers]` format. Then, via heartbeats negotiates ResourceManager its changing resource needs. After negotiation, it applies the dynamic adjustments to ensure resource consumption restraints are met. ResourceManager controls the container allocation by enforcing our plugged elasticity policy for a specific job. Before launching the container it has to construct the CLC object according to its needs which can include the allocated resource capability, security tokens, resource dependencies, environment variables, local directories. Next is to execute the task on the launched container.

To this end, our contribution is twofold: (i) An elastic and granular metering and resource scheduling mechanism for a class of MapReduce-based applications. (ii) In support of such a mechanism, we develop an elastic data telemetry (TED) framework as a basis for a multi-level resource metering model. This contributes to fine-grained resource consumption transparency and elasticity control. Our TED framework implements a layered approach to the interpretation of the time series resource usage events. TED achieves resource granular metering model in a hierarchical modeling topology. Having the metered data collected, TED highlights operational events, aggregates, and enriches them with the corresponding pricing model, then correlates the data with the associated client account and generates the billable artifact respectively. This leads to useful insights into the MapReduce job behavior for generating new allocation policies to achieve the intended performance defined in the service level agreements (SLAs).

With this motivation in mind, this chapter continues with an initial analysis to identify and elicit the requirements for the TED framework in section 7.2. With some definitive clues on how the MapReduce jobs are currently being metered and monetized, we propose a new elastic data telemetry framework (TED) to fulfill each requirement. The TED framework layered architecture together with its interacting components are detailed in section 7.3. This section is devoted to the core elements of the TED model *i.e.,* resource usage retrieval, MapReduce metering metrics, pricing kernel, a classification model for granular usage pattern interpretation, and a billing gateway to expose telemetry billable artifacts via FIX protocol. In support of our model, we have developed a primary TED framework able to handle MapReduce job metering in a Hadoop YARN cluster. The available prototype[3] is put to production by metering real MapReduce jobs. We evaluate our TED framework and numerical results will be given in section 7.4 to prove the efficiency of our model. Subsequently, section 7.5 surveys related work. Finally, section 7.6 concludes the chapter and presents an outlook on future research directions.

7.2 Resource Consumption Metering Requirements

In architecting our metering middleware we initially defined the requirements in which it will operate. The appropriate metrics for the metering methods and the jobs that are metered could vary quite significantly based on factors such as telemetry requirements and application domain context. Next, we elicit our requirements.

7.2.1 YARN Cluster Capacity Planning via Metering

YARN clusters are elastic in nature. They host MapReduce applications and adapt themselves to varying loads by elastically allocating and releasing underlying resources to map and reduce tasks. Obviously, it is an obligation for YARN providers to assure the availability of required supply. Otherwise, lack of resources results in unmet demands and poor performance, triggering the SLA violations and leads to financial consequences and penalties. YARN supply planning depends on a few factors: types of machines (Nodes), types of workload (Memory/Storage/CPU-intensive), Number of tasks (map or reduce) per node and etc. Usually count 1 core per task. If the job is not too heavy on the CPU, then the number of tasks can be greater than the number of cores. For instance: 12 cores, jobs use 75% of CPU, free task slots = 14, maxMapTasks = 8, maxReduceTasks = 6. By default, the tasktracker and datanode take each 1GB of RAM. For each task calculate `mapred.child.java.opts` (200MB per default) of RAM. In addition, count 2GB for the OS. So say, using 24GB of memory, 24-2=22GB available for our tasks – thus we can assign 1.5GB for each of our 14 tasks (14 * 1.5 = 21GB). YARN uses `yarn.nodemanager.resource.memory-mb` and `yarn.nodemanager.resource.cpu-vcores` to control the amount of *memory* and *cpu* on each node, which both are available to *maps* and *reduces.* Resource allocation plans and elasticity requirements can be enforced using these configurations.

Metering can provide valuable information about these factors to show the way that an application is used. This requires classifying the job types (CPU bound, Memory or Disk I/O bound, or Network I/O bound) into: balanced workload, compute intensive,

[3]https://github.com/soheil4TUWien/TED

I/O intensive or evolving workload patterns. Such classification can identify trends that indicate future needs such as storage and compute resource requirements. Moreover, this may indicate which resources are more utilized in map or reduce intensive job phases affecting response times. Such utilization knowledge might also influence the effort towards development of optimized storage methods, cost reduction or additional storage, like using AWS spot instances[4]. The fundamental unit of resource allocation in YARN is the *priority queue* which we will discuss later in details.

7.2.2 Metering for Granular Consumption Transparency

The level of a resource metering unit becomes an essential facet of facilitating a way of hierarchical metering and processing of usage patterns indexed by information granules. By granule metering, we mean a collection of metrics aggregated together by their functional relationship or closeness. Such granules are then formulated by adopting and leveraging a certain level of abstraction to achieve further utility. Each abstraction level is formed by grouping metrics together into semantically meaningful constructs to reflect the structure of the original data into its granular counterpart. Granular metering enables diving deeper into measuring the resource usage on the *per-DAG-flow*, *per-DAG*, *per-Job*, *per-Map* or *per-Reduce* levels. Such metering granules can be regarded as more abstract and interpretable entities in charging clients and in elasticity policy enforcement. We treat them as a scale unit. This provides users real-time visibility over their resource consumption and the ongoing money stream being paid as they go. Furthermore, it enables clients realtime application control to ensure that quality and cost constraints are met.

Meanwhile, we see the resource usage events as time-series data whereas the consumption information granulation occurs in time intervals. In this context, the duration of the map or reduce phases' run could form a scale of time granulation. As elaborated above, we identify three factors: *granule unit*, *time interval* and *metric type* to building granular representatives of usage data. The granulation mechanism involves criterion of closeness of elements, and if required, could also embrace some aspects of functional resemblance. In other words: we collect underlying usage events and elevate them into granule units in reasonable time intervals. Such granular classification of application metering enable *Pay-per-granules* enables for mapreduce-based applications where customers are billed based on their actual application resource usage and can track their ongoing costs.

7.2.3 Fine-grained Metering for Elasticity Control

Elastic metering allows fine-grained adequate resource allocation and prevents exceeding the preset resource usage and limits. By elastic metering we mean to enforce resource quotas on the cost and quality constraints. This requires a scheduled utility that observes key threshold constraints and fires the appropriate notification event to enforce the suitable elasticity control policy like *Scale-in* or *Scale-out*. To apply such control, we

[4]aws.amazon.com/ec2/purchasing-options/spot-instances/

provide a granular classification on the usage data. The schemes of granular classification are comprised of several functional steps. A crux of the scheme is shown in the following:

$$\text{Usage Events (1)} \xrightarrow[\texttt{Retrieval}]{\texttt{[key,value]}} \text{Metrics Feature Space (2)} \xrightarrow[\texttt{Aggregation}]{\texttt{Granulation}} \text{Granular}$$
$$\text{Feature Space (3)} \xrightarrow[\texttt{Association}]{\texttt{Correlation}} \text{Interpretation (4)} \xrightarrow[\texttt{Enforcement}]{\texttt{Tailored Policy}} \text{Queue Classifier}$$

Let us briefly elaborate on the essence of the successive phases of the overall metering scheme. The first step is dedicated to meaningful representation of the resource usage data to form a metric feature space. As a result of this representation, one returns a vector of numeric descriptors characterizing the time series and used in consecutive phases. This vector of `jobTaskAttemptCounters` contains a list of `[key,value]` of our job metrics. In our case we retrieve the `CPU_MILLISECONDS`, `PHYSICAL_MEMORY_BYTES` and `VIRTUAL_MEMORY_BYTES` metric values of tasks for further processing.

At any given time t in the second phase, TED provides enriched pieces of information about the MapReduce job resource usage observations. Such observations can be represented as a column-vector $\mathbf{o}_t \equiv [\,v_{t,1}\ v_{t,2}\ \ldots\ v_{t,n}]^T \in R^n$ of YARN metrics data stream values at time t. The stream of usage data can be regarded as a frequently expanding $t \times n$ matrix $\mathbf{O}_t := [\mathbf{o}_1\ \mathbf{o}_2\ \ldots\ \mathbf{o}_t]^T \in R^{t \times n}$ where the new incoming streams are added as matrix rows at each time interval t in real-time. In our YARN case, $\mathbf{O}_t$ is the measurements column-vector at t over all the metrics, where n is the length of the vector and indicates the number of hadoop metrics and t is the measurement time-stamp. These vectors represent the set of measurements obtained for the n metrics at a specific observation. In particular the rows of the matrix represent various `monitoring observations` in a given period, while the columns are the sample `values` detected for each metric during the observations.

$$\mathbf{O}_{t,n} = \begin{pmatrix} jobID_{1,1} & taskID_{1,2} & CPU_{1,3} & \cdots & Mem_{1,n} \\ jobID_{2,1} & taskID_{2,2} & CPU_{2,3} & \cdots & Mem_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ jobID_{t,1} & taskID_{t,2} & CPU_{t,3} & \cdots & Mem_{t,n} \end{pmatrix} \tag{7.1}$$

In our scenario, as shown in Equation 7.1, let $\mathbf{O}$ be a matrix representing the job's tracking data measured by metrics counters in the allocated container. For instance, `key` element of $CPU_{t,3}$ represents the CPU usage `value` of the specific $taskID_{t,2}$ of the specific $jobID_{t,1}$ at time t. This leads to granular representation of time series data.

In the third phase, a collection of information granules is constructed and positioned as belonging to granularity classes. This forms a layered approach to constructing a classification framework of granular interpretation of time-varying resource consumption events. Such interpretation abilities could help to understand the type and behavior of the job in terms of resource insensitivity. The queue classifiers positioned as the last functional module of a metering framework scheme are used to realize mapping of discovered running job types on the current job scheduling policy and its associated elasticity control strategy class labels. This mapping helps in choosing the suitable economic decisions and accordingly actions taken on the job elasticity control like moving the job to a proper queue.

Having such classifications in effect, TED manages the level of resource provisioning on various granules to a specific YARN deployment to keep the job up and running and ensure elasticity requirements are met. Next, we describe the design and implementation of our metering solution for mapreduce-based applications deployed in YARN cluster. Last, but not least, about the architecture robustness, if TED faces its own deployment issues then this may have a major impact on vendor profitability. Our approach is to schedule some background log analysis utilities like logstash[5] to detect and even restart the suspended TED instance.

7.3 TED Framework Architecture

Looking forward, figure 7.2 provides a schematic view on architecting TED's collaborating components. The next section lays out the basis for the underlying resource metering metrics together with its retrieval mechanism.

7.3.1 Usage Data Retrieval

Collecting and streaming usage data from mapreduce jobs in Hadoop needs a lightweight solution to avoid additional network I/O for the sake of performance. Each usage event contains information about the job like subscriber, timing, the result (success, failure), resource usage metrics and their values. One solution is to receive notification events via callback feature of the Hadoop. At job completion, an HTTP request will be sent to `job.end.notification.url` value[6]. Both the `JOB_ID` and `JOB_STATUS` can be retrieved from the notification URL that we supplied in Job configuration. The URL connection is fire-and-forget (FAF)[7]. We take a skeptical view of this approach since we will not receive any notification in case of task completion. Typically, when you run a map/reduce job you get the object of type `org.apache.hadoop.mapreduce.Job`. Using this object you can poll the JobTracker in a predefined interval to check its status.

The collection of monitored metrics data that are exposed by Hadoop Metrics2 system daemons are written into the time-series database to make it ready for querying and processing. We register our sinks to retrieve our desired metrics. The metering event

[5]http://logstash.net

[6]http://localhost:8080/jobstatus.php?jobId=$jobId&amp;jobStatus=$jobStatus

[7]http://www.w3.org/TR/xmlp-scenarios/#S1

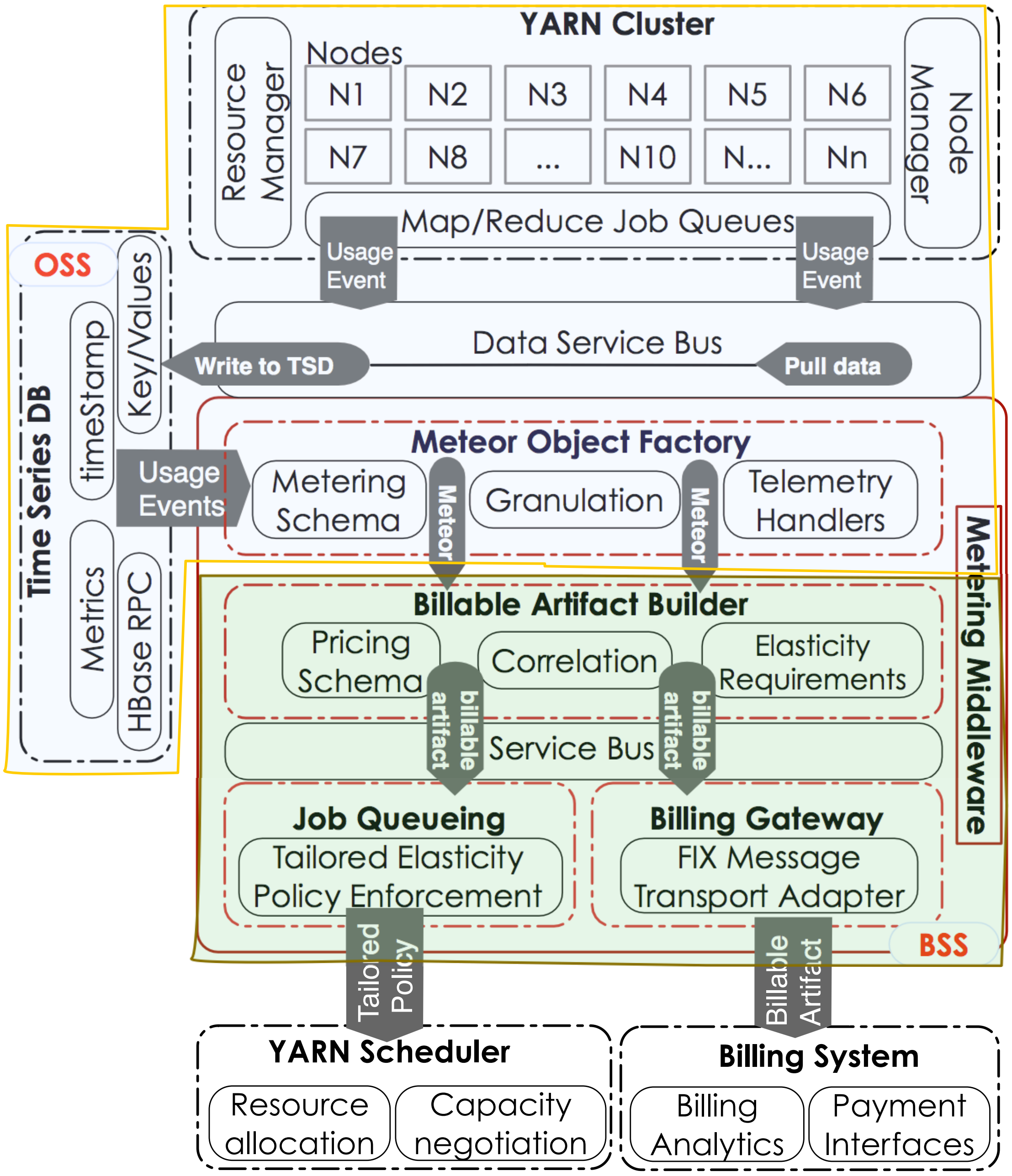


Figure 7.2: MapReduce Job Metering & Rating Middleware Architecture.

data along with its job metadata is streamed to a tree-like[8] schema in a time series database to reflect our metering granularity in hierarchical layering. The metering tree

[8]We use OpenTSDB 2.0 tree structure, a hierarchical method of organizing time-series into an easily navigable structure.

schema is a JSON object that defines the hierarchical granularity topology in a tree model. TED traverses the tree via its HTTP API endpoint for further metering processes. The root(depth: 0) of the metering tree is the YARN cluster. One depth level below root is the DAGs-flow where the workflow of mapreduce jobs are hosted at the depth of 1. Moving forward, the MapReduce DAG branch is exposed at depth 2. Next, the depth increases to 3 where the MapReduce Job branch resides. This leads to a deeper branch of depth 4 where leaves exist with the map and reduce task's granularity. Navigating to the leaves of the tree represents the actual data points for metering events. With this approach implemented, TED will receive metering events and data for the completed tasks in the form of REST calls and JSON formatted data.

7.3.2 Meteor Object Factory (MOF)

So far, we have retrieved the usage data and loaded in the time series database in a dynamically built tree-like schema to keep the granularity. Once the metering data has been accumulated, the MOF component navigates the tree branches and their leaves as an input for constructing the *Meteor* objects. The MOF parses the metering events to extract resource usage `keys` and `values`. The meteor objects are basically aggregated metering events grouped by specific granularity and a user ID. For instance, a meteor on the MR-app granularity contains all the jobs' aggregated MR-tasks resource usage into a summarized JSON object. We call this object, a Meteor which will be processed and used to charge the user. Since the user already knows the size of their coming meteor, they then can be prepared in advance for that financially, etc. A meteor object structure is driven by `[key,value]` parity. The actual meteor that we generate carries 8 elements of which 6 represent the metrics.

To take a closer look of a meteor, think of metering the usage of a service. If service metering schema indicates that the metering pattern should be based on the number of service invocations. Then, having 10 calls from a service subscriber enables the telemetry instrumentation system to collect 10 metering events and the MOF generates 1 Meteor object indicating the service was invoked 10 times with the summary of resource usage. The process of meteor construction can be carried out on a predefined frequency of data collection. Meteors are built at various level of granularity that makes it easier to interpret, discover trends, gain insights into usage and performance for future elasticity strategy/policy selection and enforcement. When construction and transmission of some meteors might have priority in terms of their influence in elasticity decision making, then the priority queue pattern is considered in its life-cycle.

7.3.3 Telemetry Handler

The Telemetry Handler (TH) frequently invokes the *Granulation* REST API to generate the *Meteors* by aggregating the usage events into granules defined in the metering schema. Each meteor represents one completed job's aggregated resource usage. Then TH observes and audits the meteors JSON objects with the elasticity requirements to detect if any threshold is hit. The proper allocation policy (Scale Up/Down) will be plugged in if any check constraint is violated.

7.3.4 Billable Artifact Builder (BAB)

The constructed meteors in the MOF component will be transmitted to the BAB component. The BAB rates and monetizes the meteors by enriching them with the associated pricing schema and the user profile into a billable artifacts. Billable artifacts are granular resource usage-centric constructs capturing financial valuation of the abstract entities like job, map, reduce, etc. They indicate the econometric of the aggregate consumption data. The enriched meteors will be correlated with the elasticity controls for future policy enforcements, allocating or releasing resources, for instance. Job subscribers will be charged based on their usage pattern indicated in their billable artifacts. BAB enables an end-to-end mapping of the operational meteors and pricing schema to expose metered and billable artifacts to the billing system. This achieves a fine-grained unit of work for metering and pricing on the fly at economies of scale. BAB measures a number of metrics (CPU, storage, memory, etc) in `[key, value]` elements embraced by the granule and exposes them as billable artifacts to billing systems. Moreover, this exposure reveals the cost incurred on currently running mapreduce-based applications. The spending meter is updated at intervals to keep users aware of their payment stream in real time.

7.3.5 FIX Billing Gateway

Financial Information eXchange (FIX)[9] protocol is an open messaging specification to streamline electronic communications among financial entities for trade allocation, order submissions, etc. FIX billing gateway offers an architecture for exposing the billable artifacts to external billing systems (BS) and keep the pricing and metering schemata updated from partner endpoints.

In our solution the metering service bus acts as the core message gateway, sending various billable artifacts and services to FIX endpoints using the built-in FIX transport adapter. The FIX message gateway connects endpoints by transforming messages to standard FIX messages using its base data dictionary and specifications. TED FIX protocol implementation is illustrated in figure 7.3. Proxy services are configured to transport billable artifacts in FIX messages to BS via message broker (*i.e.,* Apache Synapse[10]). The service bus converts FIX messages into XML which will be wrapped inside the Synapse message and sent to the BS. The FIX transport layer maintains session message correlation using message-id and correlation-id that allows the ESB to send relevant executions and acknowledgments back to the original FIX endpoint.

[9]http://www.fixtradingcommunity.org/
[10]http://synapse.apache.org/

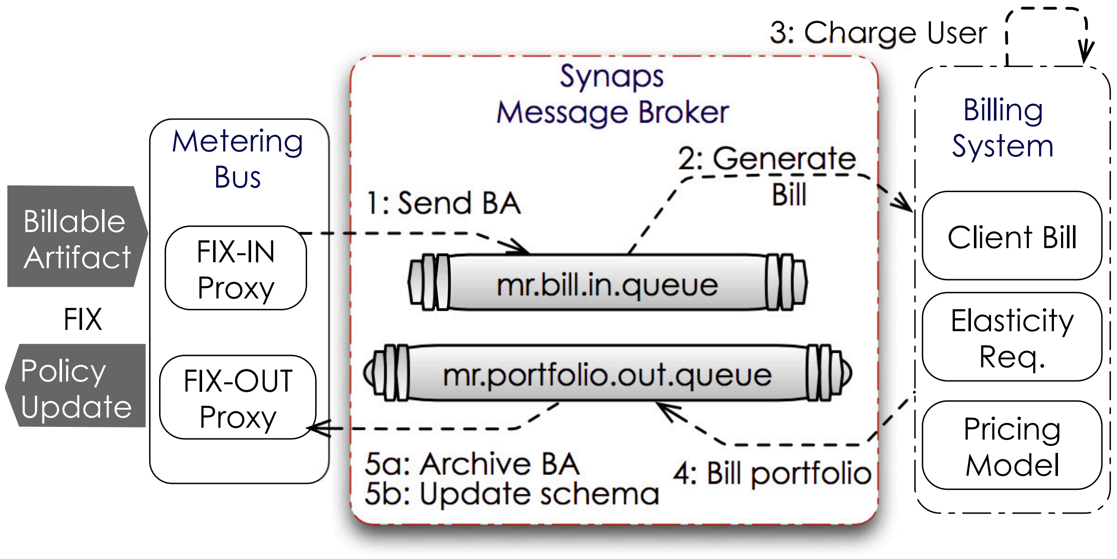


Figure 7.3: TED tunnels billable artifacts using FIX protocol.

7.3.6 TED Job Queueing

Queueing theory is modelled on how to serve many arriving jobs while having scarce resources. In YARN, a queue is a logical collection of applications with a guaranteed resource capacity. They reflect the economies of resource allocation policies. Given the job's resource requirements, the goal is to improve performance by deploying a more optimized scheduling policy to achieve economies of scale. Our tailored policy in this study is to move the application to a queue which satisfies the application launch context. It is an indication that the following metrics underlying our model have been considered in rating the target queue for the proper positioning of the application.

◇ *Queue Utilization*: (ρ_i) is the fraction of time a container i is in use (non-idle). It is calculated by total observation of busy time (T_b) over length of observation period (τ), that can be formulated in $\rho_i = \frac{T_b}{\tau}$ equation.

◇ *Queue Throughput*: (Xi) is the rate of task completion (e.g., jobs/sec) at container i. Formally, the total number of completed jobs, J at container i within period of (τ) results in $Xi = \frac{J_{vm}}{\tau}$ throughput.

◇ *MapReduce Job Size*: (S_{mr}) indicates the amount of required time to run a job on the specific CPU alone. ($E[S_{mr}]$) represents the average required time to run the job excluding the queueing time (e.g., $\frac{1}{4}sec$).

◇ *Job Average Arrival Rate*: (λ) is the average rate of the job's arrival to the queue (e.g., $\lambda = 2$ jobs/s).

◇ *Job Average Serving Rate*: (μ) is the job's average serving rate in the queue (e.g., $\mu =$

4 jobs/s $= \frac{1}{E[S_{mr}]}$).

◇ *Price of Entry & Cost of Waiting* : (P) is the job's entry price in the queue and the c indicates the cost per unit time of waiting in the queue (e.g., $P = 2$ and $c = 10\ cents/s$).

◇ *Job Migration Cost*: (M_c) contains the active job migration cost (e.g., $M_c = 50\ cents$ per/job-size). This might come with a high cost because of state (memory). Such costs will be amortized over the new queue on its remaining processing time.

◇ *User Budget*: (B_u) contains the initial user budget limit to run the job.

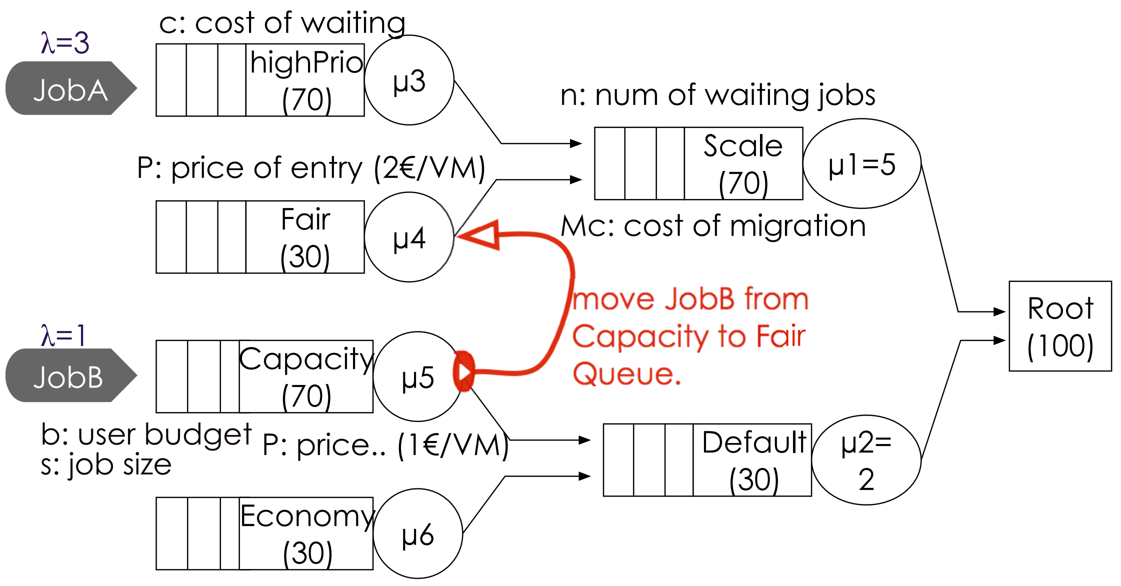


Figure 7.4: TED Hierarchical Queueing Model & Job Migration.

In our model, after the migration, we enforce the shortest remaining processing time (SRPT) priority algorithm as a serving mechanism for the migrated jobs. TED's queueing model is illustrated in Fig 7.4. In this mode, the used capacity of any parent queue is defined as the aggregate sum of used capacity of all the descendant queues recursively. the used capacity of a leaf queue is the amount of resources that are used by allocated containers of all applications running in that queue. Such a container is a unit of resource allocation across multiple resource types incorporating resource elements such as memory, CPU, disk, network etc, to execute a specific task of the application.

Algorithm 7.1 shows how applications are moved across queues in TED. The scheduling algorithm clarifies when to move which application to another queue. It is implemented and evaluated in the following section. The algorithm applies only for running jobs which are submitted on a specific queue.

Algorithm 7.1: TED scheduling algorithm for running jobs

```
1: procedure MoveToQueue($AppID, targetQueue$)
2:     if (AppStatus == "Running"
3:     && AppQueue == "Cloudera"){
4:     mapTasks[] ← list of maps of current Job
5:     reduceTasks[] ← list of reduces of current Job
6:     //check for the first completed map task
7:     for (each task in mapTasks)
8:     if (state of task equals "Succeeded")
9:     completedMapTaskId←currentMapTaskId;
10:    Compute $\rho_i$ and $Xi$ to classify highPriority queues
11:    if (completedMapTaskId ≠ null){
12:    // read the completed map task information from tsdb
13:    CPU ← store CPU usage of map task (ms);
14:    // Store HEAP & VMEM & PMEM values used by mapper
15:    costPerMap = CPU × targetQueue_cpuCost
16:    // plus the sum of HEAP, VMEM and PMEM;
17:    //estimate cost of job on the target higherPriority queue
18:    totalEstimatedCost=costPerMap×num.map-Tasks +
   costPerMap×num.reduceTasks+P;
19:    if (totalEstimatedCost < user_budget)
20:    app.setQueue(targetQueue); }
21:    }
```

7.4 Model Evaluation

Now, we present results from our real-world observations that show the efficiency of our model. We have implemented two threads; one for streaming the usage metrics data to tsdb and the other one for creating the meteors, cost estimations and migrating the job to the target queue. As a real job, we executed the YARN Pi example which computes the Pi value with the given precision with two configurations. In the first case, we have run the Pi job on 60 mappers with 30 samples per map. As for the environment set up, the Fair scheduler of YARN is in place and we have two queues of lowPriority (weight =1) and highPririty (weight = 2). The observations on CPU consumption growth is trending over queue change in time. The aggregate results imply utility and are summarized in Fig 7.5.

Taking these results together, four points stand out in this evaluation. First, the job was running in the first queue for about 25 seconds. Second, TED decides to migrate the job to a higherPriority queue after 25 seconds of jobs execution in which only 4 maps have been executed. Third, the migration takes 5 seconds and the job continues to run in the new queue at the starting time of 22:58:30. Finally, the first 4 maps were executed in 15 seconds while after migration the remaining 56 maps together with reduces executed in

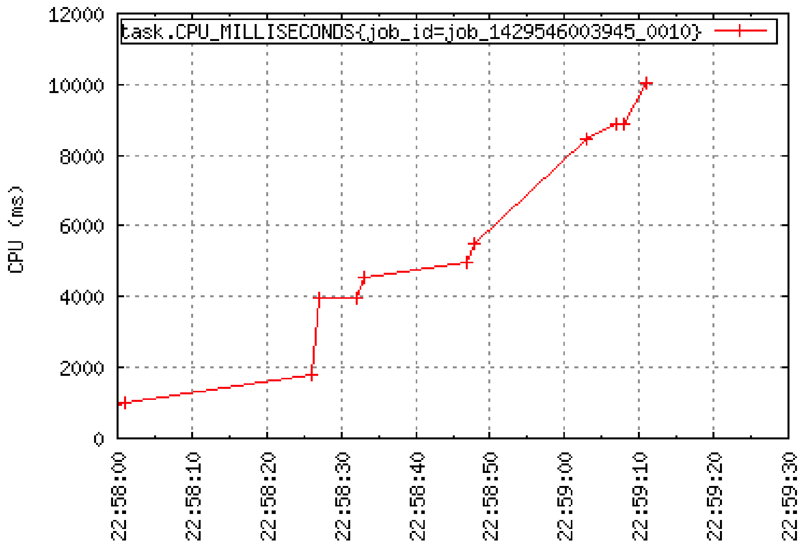


Figure 7.5: YARN Pi example with 60 mappers and 30 samples (OpenTSDB *sum* diagram).

40 seconds in the new queue.

Running the Pi with the second configuration results in Fig 7.6. It also took almost 4 maps to enforce the migration to the new queue at 02:33:00 which took 3 minutes to move to the new queue due to the larger size of the job. Having the migration in place, we can observe that the job is utilizing more resources to be finished. Since we had the SRPT policy in place, at the end of job, we see that it has the highest priority to be finished sooner. Results presented above are convincing enough to lead us to ascertain that in the highPriority queue the allocated memory and CPU were considerably higher to execute the job faster than the time it was submitted to lowPriority queue.

The empirical evidence observed in this study suggests that our consumption-based pricing model for MapReduce jobs provides statistically and economically important insights into financial behavior of an underlying resource usage model. TED enables applications to implement "elasticity-aware policies" to satisfy their resource requirements. If an associated job migration policy among queues is triggered, the allocation is leveled to the pre-configured queue capacity dynamically. This capacity leveling keeps the amount of resource allocation within the range (minSize and maxSize) of an intended resource allocation.

7.5 Related Work

To the best of our knowledge, this is the first study that leverages the metering to the data processing domain. Meanwhile, there is some commendable research regarding the cloud service usage metering. Elmsroth et al.[78] proposed a loosely coupled architecture

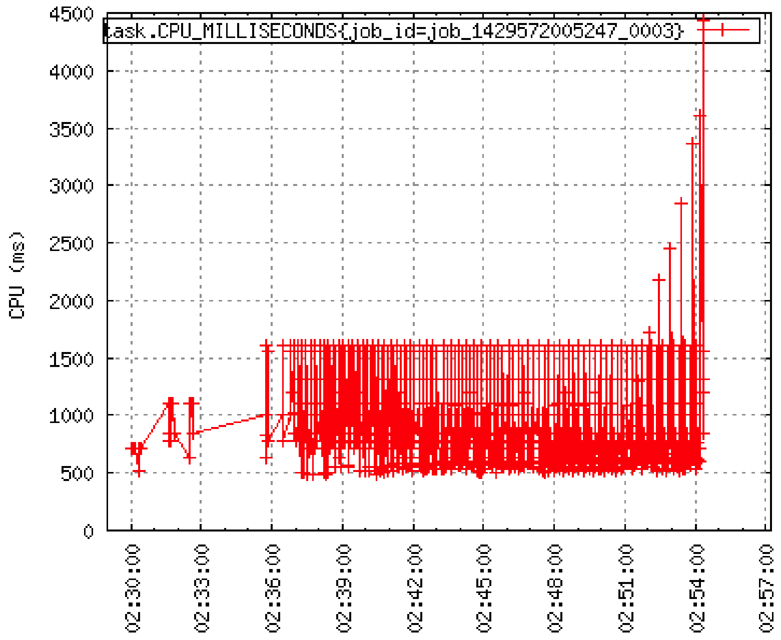


Figure 7.6: YARN Pi example with 300 mappers and 100 samples (OpenTSDB *minmax* diagram).

solution for an accounting and billing system for use in the RESERVOIR[79] project. In a separate work, Prasad et el.[80] described the usage of open source tools such as OpenTSDB, Hbase and Hadoop to store time series data and perform analytics on the time series data to get useful insights related to power consumption and get the result in pictorial format, essentially a graph. These papers do not provide any implementation or evaluation of the proposed architecture. There are some alternatives that propose billing and metering solutions, Narayan et al.[81]. Petersson[82] describes cloud metering and billing solution. Naik et al.[83] proposed a solution for metering of services delivered from multiple cloud providers. They incorporate the cloud service broker together with a metering control system to report metered data at configurable intervals. Their solution is developed and deployed as a plugin for IBM SmartCloud Enterprise. None of these solutions address mapreduce-based application metering and is not applicable to a cloud-based data processing framework like the Hadoop environment.

In relation to our approach, the SequenceIQ [11] project brings SLA policy based auto-scaling to Hadoop YARN. Their project is quite established and now is part of Hortonworks. It is built on top of YARN schedulers, allows to associate SLA policies to individual applications. It monitors application progress and apply scaling policies based on their CPU and memory usage. In contrast to all the noted related work, our TED middleware addresses granular metering of mapreduce jobs while considering the clients cost constraint regarding their jobs resource usage.

[11]http://sequenceiq.com

7.6 Conclusion

We have presented an elastic data telemetry system that enables granular metering and automatic control of MapReduce applications due to their current behavior and preset configurations. TED is designed to enhance the resource utilization using YARN cluster queueing system. It observes the running job's progress, interprets its cost and quality behavior, audits the predefined job requirements. Then it generates a tailored resource allocation policy with regard to capacity constraints and moves the job to the potential queues to scale in or up.

So far, the authors offered and implemented a solution for the YARN environment which provides a data processing telemetry solution for mapreduce-based applications. As an outlook, our future work includes further extension to the TED model to formulate intuitions for best priority queue selection in multi-queue systems. We then move on to analyzing trees of queues with trendy metrics like *slowdown*, *starvation* and *fairness* for a workflow of jobs.

CHAPTER 8

Diameter of Things

8.1 Introduction

Utility computing[74] is an evolving facet of ubiquitous computing that aims to converge with emerging Internet of Things (IoT) infrastructure and applications for sensor-equipped edge devices. The agility and flexibility to quickly provision IoT services on such gateways requires an awareness of how underlying resources as well as the IoT applications are being utilized as metered services. Such awareness mechanisms enable IoT platforms to adjust the resource leveling to not exceed the elasticity constraints such that stringent QoS is achievable.

The quest for telemetry of the client's IoT application resource usage becomes more challenging when the job is deployed and processed in a constrained environment. Such applications collect data via sensors and control actuators for more utilization in home automation, industrial control systems, smart cities and other IoT deployments. In this context, telemetry enables a *pre-paid* and *pay-per-use* economic models via utility-based pricing model through metered data to achieve more financial transparency for resource-constrained applications.

Metering measures rates of resource utilization via metrics, such as number of application invocations, data storage or memory usage consumed by the IoT service subscribers. Metrics are statistical units that indicate how consumption is measured and priced. Furthermore, metering is the process of measuring and recording the usage of an entire IoT application topology, individual parts of the topology, or specific services, tasks and resources. From the provider view, the metering mechanisms for service usage differ widely, due to their offerings which are influenced by their IoT business models. Such mechanisms range from usage over time, invocation-basis to subscription models. Thus, IoT application providers are encouraged to offer reasonable pricing models to monetize the corresponding metering model.

To fulfill such requirements, we have incorporated and extended the Diameter base protocol defined in RFC6733[1] to an IoT domain. There are several established Diameter base protocol applications like Mobile IPv4 [84], Credit-Control [85] and Session Initiation Protocol (SIP) [86] applications. However, none of them completely conforms to the IoT application metering models.

The current accounting models specified in the Diameter base are not sufficient for real-time resource usage control, where credit allocation is to be determined prior to and after the service invocation. In this sense, the existing Diameter base applications do not provide dynamic metering policy enforcement in resource and credit allocations for *pre-paid* IoT users. Diameter extensibility allows us to define any protocol above the Diameter base protocol. Along with this idea, in order to support real-time metering, credit control and resource allocation, we have extended the Diameter to Diameter of Things (DoT) protocol by adding four new types of servers in the AAA infrastructure: DoT Provisioning server, DoT Resource control server, DoT Metering server and DoT Payment server. Further details regarding the aforementioned entities will be discussed in a later section of DoT architecture models. In summary, our main contribution is the specification of an extended Diameter base protocol to an IoT application domain. This contributes to fully implementing a real-time policy-based telemetry and resource control for a variety of IoT applications. Our protocol supports both the *pre-paid* and cloud *pay-per-use* economic models.

With this motivation in mind, this chapter continues in section 8.2, with a brief review on how the Diameter base protocol functions. Next, we introduce the terms and preliminaries defined in this study at section 8.3. With some definitive clues on the Diameter architecture, we propose Diameter of Things (DoT), an extension to the Diameter on how IoT applications are to be metered and monetized. The DoT framework layered architecture together with its interacting entities are detailed in section 8.4. In support of our model, we have defined a DoT-based IoT application topology and its associated hybrid metering policies in section 8.5. This lets the application telemetry policies vary independently from clients as well as applications that use it. To enable this, DoT performs several interrogations which are detailed in section 8.6. Next, in section 8.7, we express formally the DoT application transaction models to achieve better telemetry control over computing resources. Moving forward, DoT commands are then described in section 8.8. Section 8.9 describes set of commands that are extended or reused from Diameter based protocol. In section 8.10 we introduce the DoT protocol Attribute-Value pairs (AVPs). Section 8.11 then explores AVPs used in each command in-depth and defines the mandatory AVPs for each DoT command. DoT state machines are provided in 8.12. Subsequently, section 8.13 surveys related work. Finally, section 8.14 concludes the chapter and presents an outlook on future research directions.

[1]https://tools.ietf.org/html/rfc6733

8.2 The Utility of Diameter

The Remote Authentication Dial In User Service (RADIUS) [87] as per RFC2865 is a simple but most deployed protocol which provides network access control using client/server Authentication, Authorization and Accounting (AAA) model. The IETF[2] has standardized the Diameter base protocol [88] as an enhanced version of RADIUS providing a flexible peer-to-peer operation model. It is featuring intermediary agents (Relay, Proxy, Redirect and Translation) and capabilities negotiation among servers. It enables reliable transport layer using TCP or SCTP connection. The Diameter peer connections are

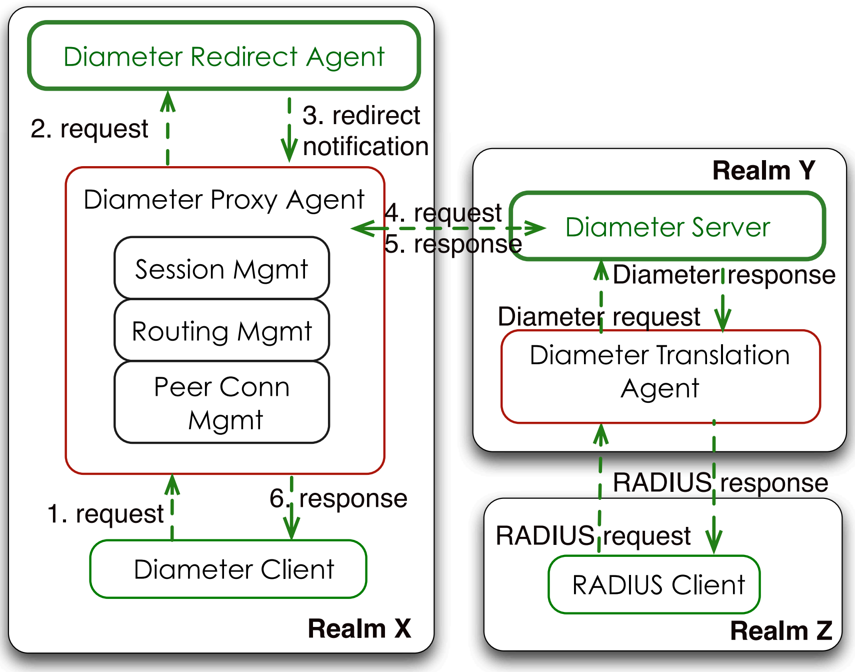


Figure 8.1: Diameter base protocol architecture

ensured using a Keep-alive mechanism. It also supports the dynamic peer discovery and configurations in a valid session. Such specifications are achieved using set of commands and Attribute-Value-Pairs (AVPs) by collaborating and negotiating peers. Diameter has the concept of "applications", which is entirely missing in RADIUS. The protocol is enriched with a globally unique application ID. These applications benefit from the general capabilities of the Diameter base protocol while defining their own extensions on top of the base.

Some of the main features offered by Diameter are dynamic routing based on Realm, session management, accounting, agent support. It is based on Peer-To-Peer architecture as illustrated in Figure 8.1, in a way that each Diameter node can behave as either a

[2]Internet Engineering Task Force (https://www.ietf.org)

client or server based on the current deployment model. Diameter nodes can be of the type of Diameter client, Diameter server and Diameter agent. Diameter client is the node that receives the user connection request. Diameter server is the one serving the request e.g. performing user authentication based on provided information. Diameter agents themselves divide into four types of Relay, Proxy, Redirect and Translate agents. A relay agent is used to forward messages to other Diameter nodes based on the information provided in the message. Proxy agent can also act like a relay agent with the extra functionality of policy enforcement implementation via message content modification. Redirect agents act as a centralized configuration repository by returning information necessary for Diameter agents to communicate directly with another node. Translation agents provide translation between two distinct AAA protocols.

8.3 DoT Preliminaries & Terms

In this section we present basic conventions, terms together with their definitions considered in the DoT protocol.

◇ *Diameter of Things (DoT)*: DoT implements a mechanism to provision IoT deployment units, control resource elasticity, meter usage, and charge the user credit for the rendered IoT applications.

◇ *IoT Microservice*: A fine-grained atomic task performed by an IoT service on a device.

◇ *IoT Application Topology*: It contains the composition of hybrid collaborating IoT microservices to meet the user's request. The topology is packages with the elasticity requirements and constraints (hardware or software) which will dictate our schedule and credit allocation within the runtime environment.

◇ *Metering Server*: A DoT Metering server performs real-time metering and rating of IoT applications deployment.

◇ *Metering Agent*: The agent transfers the metered values to the Metering server via tiny tokens.

◇ *Provisioning Server*: Provisioning server refers to initial configuration, deployment and management of IoT applications for subscribers. It also deals with ensuring the underlying IoT device layer is available to serve.

◇ *Payment Server*: The micropayment transaction charges subscribers upon relatively small amounts for a unit of usage. It basically transfers a certain amount of trade in the payWord or microMint micropayment schemes[89]. In the payWord scheme a payment order consists of two parts, a digitally signed payment authority and a separate payment token which determines the amount. A chained hash function, is used to authenticate the token. The server then calculates a chain of payment tokens or paychain. Payments are made by revealing successive paychain tokens.

◇ *Rating*: The process of giving price to an IoT application usage events. This applies to service usage as well as underlying resource usage.

◇ *Resource-control Server*: Resource control server implements a mechanism that interacts in real-time with a resource and credit allocation to an account as well as the IoT application. It controls the charges related to the specific IoT application usage.

◇ *One-cycle event*: It indicates a single-request-response message exchange pattern, which one specific service is invoked by one consumer at a time, while no session state is maintained. One message is exchanged in each direction between requesting and responding DoT nodes.

8.4 DoT Architecture Models

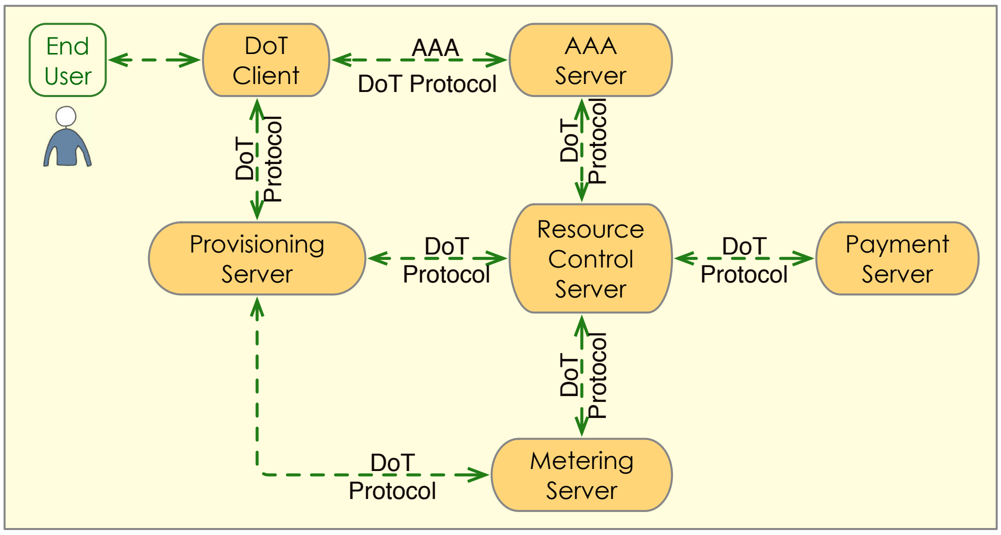


Figure 8.2: Typical Diameter of Things (DoT) application architecture

Figure 8.2 illustrates a schematic view on collaborating components of our proposed DoT architecture. It contains of a DoT client, Provisioning server, Resource control server, Metering server, and a Payment server.

As the end user defines and composes an IoT application, the request is forwarded to the DoT client. The DoT client submits the composed application to the DoT infrastructure which determines possible charges, verifies user accounts, controls the resource allocation to the application, meters the usage, and finally generates a bill and deducts the corresponding credit from the end user's account balance.

DoT client contacts the AA server with the AA protocol to authenticate and authorize the end user. When the end user submits the IoT application topology graph, the DoT client contacts the Provisioning server to submit an application topology. Afterwards, the Provisioning server contacts the Resource control server with information of required resource units. The Resource control server reserves the resources that need to be allocated to the service. As the user's credit is locked in *pre-paid* model for the application provisioning, the DoT client will receive the grant resource message. Then it informs the end user that the request has been granted. As soon as the IoT application is deployed and instantiated, the submitted topology is registered to the Metering server for telemetry and credit control purposes.

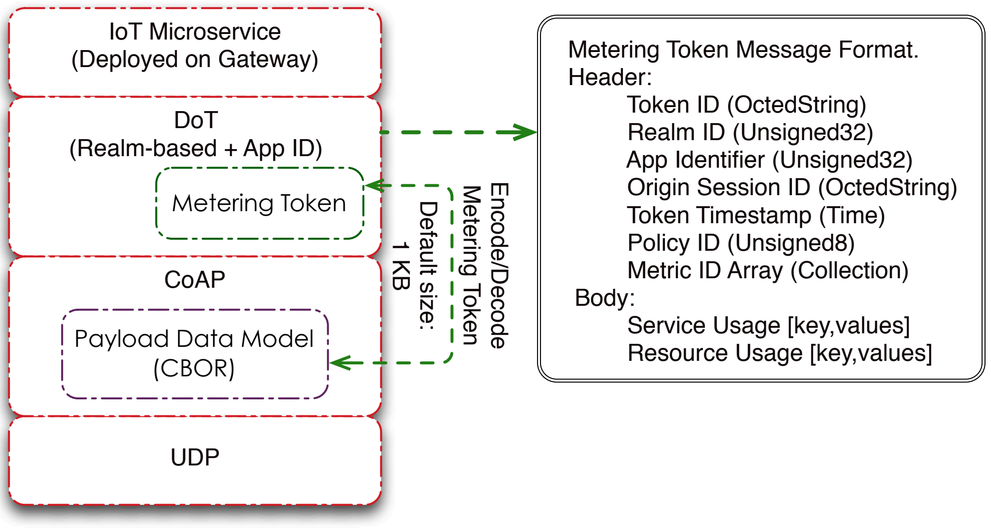


Figure 8.3: Typical DoT metering token structure

The Metering server is responsible to perform the metering transaction according to the submitted topology and meter the services by calling the Metering agent of each service in the chain. Metering transactions will remain running until the termination request is sent from DoT client to the Provisioning server. After receiving a termination request, the Resource control server releases the resource and sends the billable artifacts related to the user usage to the Payment server. The Payment server, then, invokes the payment transaction and deducts credit from the end user's account and refunds unused reserved credit to the user's account.

In DoT application architecture, metered values are transfered via tokens. The metering token message attributes as shown in Figure 8.3 must be supported by all DoT implementations that conform to this specification. The CBOR[3] message format is considered for the metering tokens transmission as it can decrease payload size compared to other data formats.

8.5 DoT-based IoT Application Overview

The DoT application defined in this specification implements flexible metering policy as well as the definition and constraints of the application topology.

[3]The Concise Binary Object Representation: http://cbor.io

8.5.1 DoT-based Application Topology

The main responsibility of the Provisioning server is the actual provision of the requested IoT application package. It contains the topology of collaborating IoT microservices to meet the user's request. Each IoT microservice is predefined with a detailed specification such as its ID, constraints and various usage patterns and policies. For instance, as defined in Listing 8.1 they can be advertised with diverse pricing models due to the event-based or time-based patterns for specific subscribers. The IoT microservices elasticity requirements and constraints (hardware or software resources) are also defined in the topology which will dictate our schedule and credit allocation within the runtime environment. The end user's request can be received in the form of a JSON (or more precisely JSON-LD [4]) object. It contains user information as well as the user requirements in terms of IoT composite application topology and its specification, to realize the intended behavior.

In order to flexibly describe the application topology, its composite architecture and deployment specification, we employ the DIANE[90] approach in utilizing the MADCAT[91] *Technical* and *Deployment Units*. After receiving the request, the Provisioning server generates a dependency graph of the application topology complying with its specification.

The Dependency graph displays dependencies between different microservices which are requested to be in topology. In the DoT protocol, this dependency graph is used in forming the transaction model for metering the IoT deployment unit. The dependency graph is a directional graph where each node of the graph represents an available microservice in the service package registry. Similarly, each edge of the graph shows dependencies between two microservices (two nodes). The edge has a direction that shows the execution order of microservices involved in this edge. Additionally, each edge has a label which shows the policy to be in effect for this connection. The Resource control server realizes such metering policies using a predefined incident matrix. This incident matrix represents the metering policies for our directed acyclic graph (DAG) of IoT services. The metering policy incident matrix P_t is a $n * m$ matrix of P_{ij} policies, where n is the number of nodes (vertices) and m is number of lines (Edges). In the cell of N_i and V_j, the P_{ij} indicates the rate of call per granted unit (time & events). It enables each service to invoke its neighbor with the attached policy. Therefore, when a client sends a request containing a tailored IoT application topology, the Resource control server is able to rate the request based on the enforced metering policies of time (duration) and event-based usage patterns.

Listing 8.1: An excerpt of an IoT application policy in a JSON object

```
{
   "microserviceID":"01",
   "microserviceName":"getTemperature",
   "uri":"getTemperature.py",
   "execute":"python getTemperature.py",
   "constraints":{
      "runtime":"python 2.7",
```

[4]JSON for Linking Data: http://json-ld.org

```
        "memory":"..."
    },
    "policies":[
        {
            "policyID":"PL_01_01",
            "cost":"$2/week",
            "desc":"time mode — $2 per week"
        },
        {
            "policyID":"PL_01_02",
            "cost":"0.01cent/invoke",
            "desc":"event mode — 0.01cent per invoke"
        },
        {
            "policyID":"PL_01_03",
            "cost":"$1",
            "desc":"subscription fee"
        }
    ]
}
```

8.5.2 DoT-based Metering Plans

The metering plans define the allocation mechanism for granting the required resource units to an IoT application/constituent microservices. It is an indication that the following assumptions underlying our IoT telemetry solution has been considered for the proper positioning of DoT protocol. The IoT applications are advertised with an associated charging plans. In case of cloud-based model, there may be a subscription fee for *pay-per-use* plans. The cost of obtaining such plans is known as the plan's "premium" which is the price that is calculated and offered in the subscription phase by the provider. The estimation of the plan's premium is out of the scope of the DoT protocol. The plan indicates the composed services pricing schema and comes in two models of predefined as well as customized. The plan will be built to be consistent with the composed application topology in the rate setting.

The subscribed metering plan, as shown in Listing 8.2, indicates:

(i) the price of granted units for every IoT application constituent microservice.

(ii) maximum allowed usage unit for each constituent microservice in case of *pre-paid* model. For instance, 50 hours for Humidity sensor and 100 invocations for Chiller On/Off actuator.

(iii) the resource Used Unit Update (U3) frequency for all associated units which are defined in the provider's plan.

(iv) the manual/automatic payment configuration. So that the provider can handle the payment transactions automatically while informing the user.

(v) container instance fee, which is the fee that user pays for underlying resource usage. Instance usage can be time based or underlying resource-usage based which is defined by the provider's policy.

(vi) finally, subscription time for *pay-per-use* model and subscription fee for *pre-paid* model.

Listing 8.2: A sample subscription plan in JSON-LD format

```
{
    "@context": "http://dchc.dot.protocol/",
    "@type": "MeteringPlan",
    "name": "DCHC/MeteringPlan",
    "artifact-uri": "https://github.com/soheil4TUWien/DoT/DCHC/Plan",
    "version": "1.0.0",
    "language": "JSON",
    "subcription": {
       "prepaid": {
            "fee" : {
                 "value": "200",
                 "unit": "Dollar"
            }
        }
    },
    "microservices": [
        {
            "name": "DCHC/Humidity",
            "type": {
                "value": "time-based",
                "unit": "second"
            },
            "maxunit": "1000",
            "U3": "10",
            "price": {
                "service": {
                    "value": "0.1",
                    "unit": "Dollar"
                },
                "resource": [
                    {
                        "cpu-usage": {
                            "value": "0.01",
                            "unit": "second"
                        },
                        "memory-usage": {
                            "value": "0.005",
                            "unit": "byte"
                        },
                        "network-received": {
                            "value": "0.005",
                            "unit": "byte"
                        },
                        "network-sent": {
                            "value": "0.005",
                                "unit": "byte"
                            },
                            "filesystem—usage": {
                                "value": "0.007",
                                "unit": "byte"
                            },
                            "uptime": {
                                "value": "0.2",
                                "unit": "ms"
                            } } ] }
            },
                    {
                    "name": "DCHC/Temperature",
                      ...
                    } ] }
```

8.6 DoT Interrogations

For a Hybrid DoT, four main interrogations are performed for a well-functioning protocol. The first interrogation is called Initial Identification (II) which basically deals with clients' identification, for instance the user authentication and authorization processes. The second interrogation is Request Realization (RR) that aims to provision the clients' IoT applications as well as scheduling their resource allocation upon agreed terms and subscribed plan. The third interrogation is called Telemetry Transmission (TT) that deals with metering the running services as well as the granted units usage data transmission for charging purposes. The final interrogation, entitled Value Verification (VV), ensures value generation and delivery to the interested stakeholders. The hybrid metering is carried out in main DoT sessions which hold globally unique and constant Session-IDs. The whole DoT-based metering life-cycle including the II, RR, TT, and VV interrogations are presented in Figure 8.4 and Figure 8.5 for *pre-paid*, as well as *pay-per-use* models.

8.6.1 Initial Identification (II)

The end user subscribes the application as well as the chosen plan to the DoT client. The DoT client submits the IoT deployment units to the Provisioning server to ask for the required resource units it needs to run. In this case, the Provisioning server queries for resources (including underlying resources and credit allocation) from Resource control server. The Resource control server is responsible for the device resource reservation. It also keeps track of user credit fluctuations.

In this phase, the end user requirements are modeled into an application topology using a directed acyclic graph. This graph can connect various IoT microservices available in diverse usage units. The deployment of such hybrid applications will result in one global constant Session-ID followed by related sub-session-IDs as well as transaction-IDs. Note that the IoT application might send a (re)authorization request to the AA server to establish or maintain a valid DoT session. However, this process does not influence

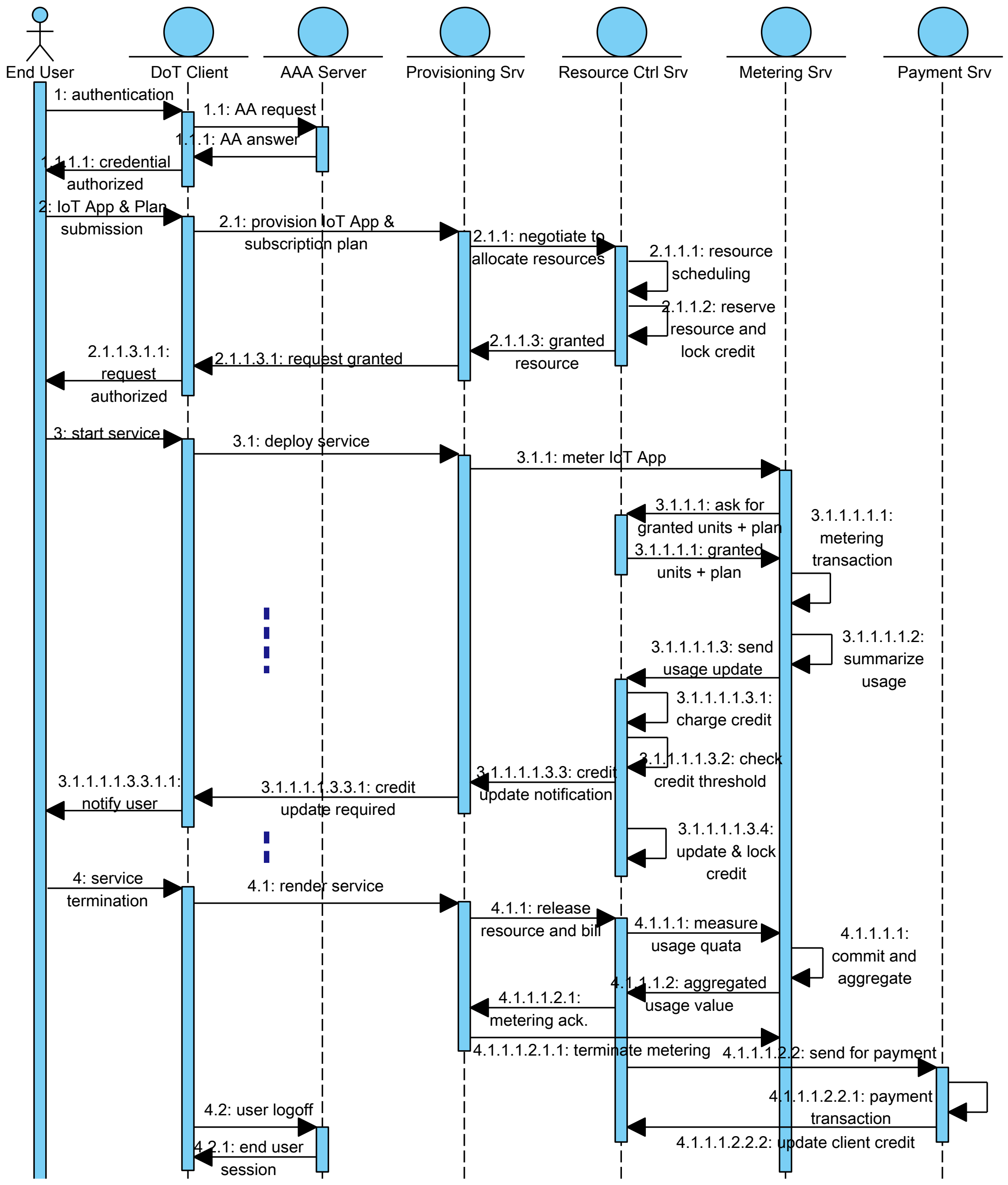

End User
DoT Client
AAA Server
Provisioning Srv
Resource Ctrl Srv
Metering Srv
Payment Srv
1: authentication
1.1: AA request
1.1.1: AA answer
1.1.1.1: credential authorized
2: IoT App & Plan submission
2.1: provision IoT App & subscription plan
2.1.1: negotiate to allocate resources
2.1.1.1: resource scheduling
2.1.1.2: reserve resource and lock credit
2.1.1.3: granted resource
2.1.1.3.1: request granted
2.1.1.3.1.1: request authorized
3: start service
3.1: deploy service
3.1.1: meter IoT App
3.1.1.1: ask for granted units + plan
3.1.1.1.1: granted units + plan
3.1.1.1.1.1: metering transaction
3.1.1.1.1.2: summarize usage
3.1.1.1.1.3: send usage update
3.1.1.1.1.3.1: charge credit
3.1.1.1.1.3.2: check credit threshold
3.1.1.1.1.3.3: credit update notification
3.1.1.1.1.3.3.1: credit update required
3.1.1.1.1.3.3.1.1: notify user
3.1.1.1.1.3.4: update & lock credit
4: service termination
4.1: render service
4.1.1: release resource and bill
4.1.1.1: measure usage quata
4.1.1.1.1: commit and aggregate
4.1.1.1.2: aggregated usage value
4.1.1.1.2.1: metering ack.
4.1.1.1.2.1.1: terminate metering
4.1.1.1.2.2: send for payment
4.1.1.1.2.2.1: payment transaction
4.1.1.1.2.2.2: update client credit
4.2: user logoff
4.2.1: end user session

Figure 8.4: The sequence of DoT interrogations in pre-paid model to enable hybrid metering

the credit allocation that is streaming between the DoT client and Provisioning server, as it has already been authorized for the whole transaction before.

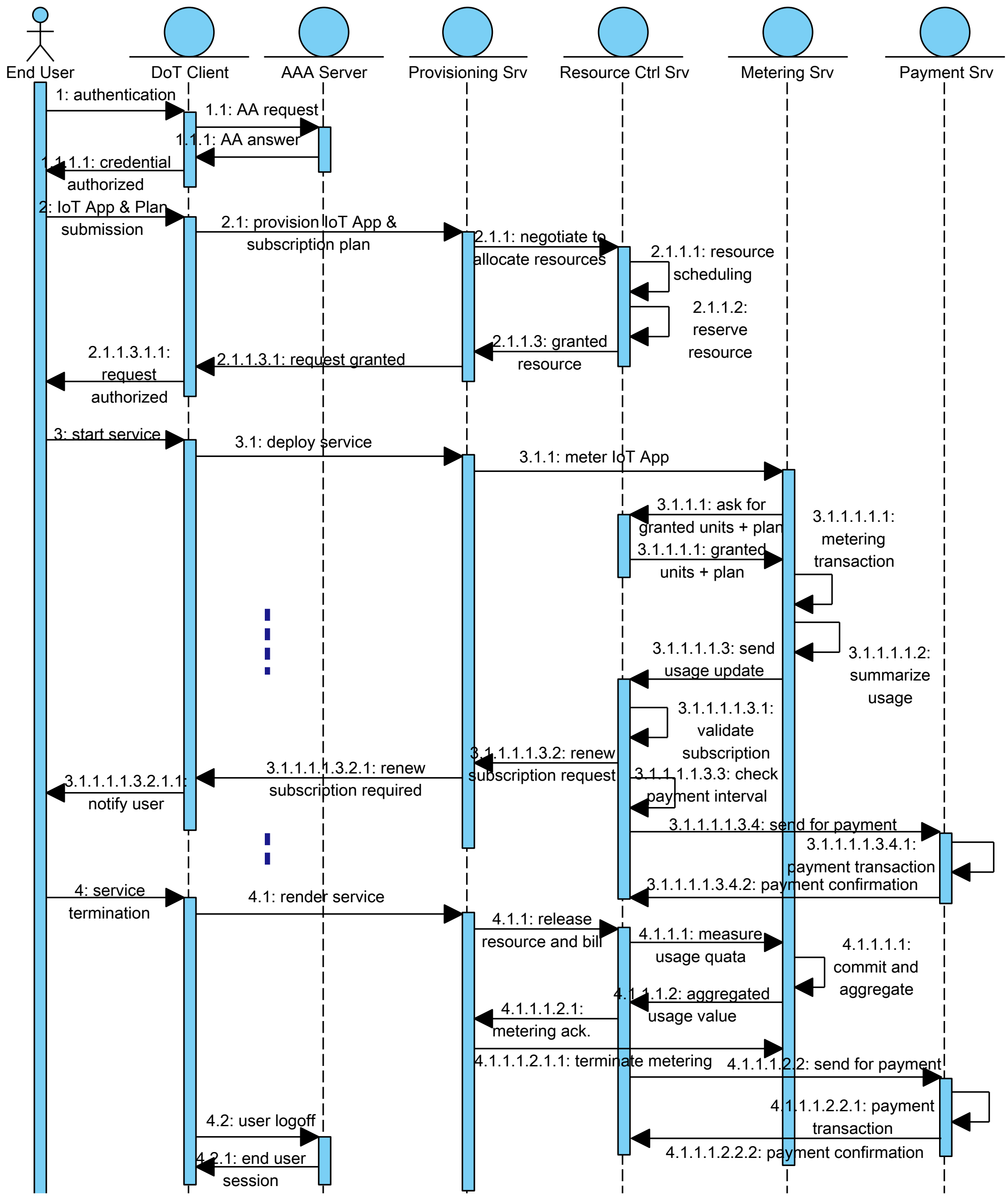

End User
DoT Client
AAA Server
Provisioning Srv
Resource Ctrl Srv
Metering Srv
Payment Srv
1: authentication
1.1: AA request
1.1.1: AA answer
1.1.1.1: credential authorized
2: IoT App & Plan submission
2.1: provision IoT App & subscription plan
2.1.1: negotiate to allocate resources
2.1.1.1: resource scheduling
2.1.1.2: reserve resource
2.1.1.3: granted resource
2.1.1.3.1: request granted
2.1.1.3.1.1: request authorized
3: start service
3.1: deploy service
3.1.1: meter IoT App
3.1.1.1: ask for granted units + plan
3.1.1.1.1: granted units + plan
3.1.1.1.1.1: metering transaction
3.1.1.1.1.2: summarize usage
3.1.1.1.1.3: send usage update
3.1.1.1.1.3.1: validate subscription
3.1.1.1.1.3.2: renew subscription request
3.1.1.1.1.3.2.1: renew subscription required
3.1.1.1.1.3.2.1.1: notify user
3.1.1.1.1.3.3: check payment interval
3.1.1.1.1.3.4: send for payment
3.1.1.1.1.3.4.1: payment transaction
3.1.1.1.1.3.4.2: payment confirmation
4: service termination
4.1: render service
4.1.1: release resource and bill
4.1.1.1: measure usage quata
4.1.1.1.1: commit and aggregate
4.1.1.1.2: aggregated usage value
4.1.1.1.2.1: metering ack.
4.1.1.1.2.1.1: terminate metering
4.1.1.1.2.2: send for payment
4.1.1.1.2.2.1: payment transaction
4.1.1.1.2.2.2: payment confirmation
4.2: user logoff
4.2.1: end user session

Figure 8.5: The sequence of DoT interrogations in pay-per-use model to enable hybrid metering

8.6.2 Request Realization (RR)

The Resource control server analyzes the IoT service and allocates resources to the requested service. It also considers the subscription time/fee in order to notify user when this value elapses.

When the new usage update arrives at the Resource control server, it charges the credit based on the usage summary received from the Metering server. It also validates subscription by checking the status of credit (as in *pre-paid* model) or elapsed time (in *pay-per-use* model) against a certain threshold. Upon reaching the threshold, the Resource control server sends an update notification request to the user. DoT protocol makes it possible to define a default action for this purpose. This default action can be to perform update automatically or to ask user to take the desired action.

8.6.3 Telemetry Transmission(TT)

As soon as the end user sends the Start service request to the DoT client, the DoT Client asks the Provisioning server to initiate the service and start monitoring and metering processes. In this regard, the Provisioning server submits the IoT application to the Metering server and asks to establish a metering mechanism for the newly opened session.

Having an IoT application deployed, the Metering server monitors the real usage of each service element including service usage and resource usage at a certain frequency rate called Unit Usage Update (U3). The U3 frequency determines the rate of sending updates regarding unit usage of each service. It is independent of the type of service. For example in case of *pre-paid* model, the U3 set to 25%, implies that for a time-based microservice with granted units of 100 minutes, the usage update should be provided every 25 minutes; and for an event-based microservice with granted units of 100 invocations, the usage update will be sent after every 25 invocations. However, the U3 rate for *pay-per-use* model is independent of individual services. It merely defines the intervals, within which the usage update from all microservices should be provided, e.g. every one hour.

To make it more clear, after identification of an IoT application to the Metering server, the Metering server sends a request to the Resource control server asking for the amount of granted units as well as the plan, which includes the U3 value for each microservice as defined by the provider. The U3 values are then provided to the Metering agents, as the Metering server starts the metering transaction.

During deployment of an IoT application, each Metering agent meters the actual resource and service usage of its associated/assigned microservice. If the actual service usage value reaches an integer multiples of U3 for a *pre-paid* plan, or if the U3 interval elapses for a *pay-per-use* model , the Metering agent will send a usage notification message to the Metering server. The Metering server then uses these feedbacks to gain a realistic perception of the usage of each microservice and to charge the user credit accordingly.

Next, if the application usage of a microservice reaches the threshold value for the *pre-paid* model, or if the subscription time of the service is close to expiry in case of *pay-per-use* model, the Metering Server informs the Resource control server issuing a resource-update request for the service. For instance, when the actual usage of a certain microservice reaches a certain threshold, e.g. more than 70% , Metering server informs the Resource control server. This contributes to a continuous and consistent service delivery. The detailed flow of TT phase is presented in Figure 8.6.

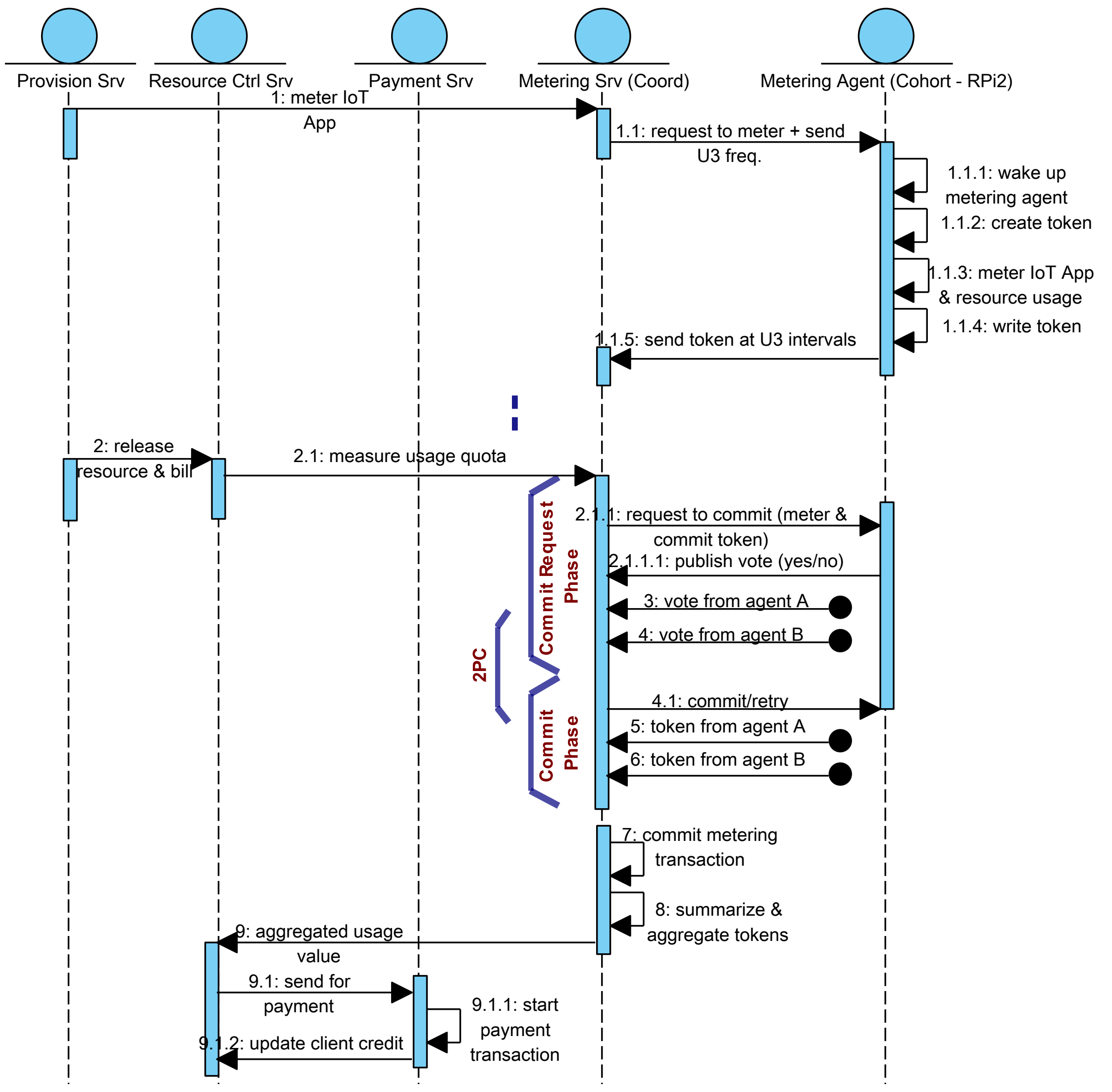

Provision Srv
Resource Ctrl Srv
Payment Srv
Metering Srv (Coord)
Metering Agent (Cohort - RPi2)
1: meter IoT App
1.1: request to meter + send U3 freq.
1.1.1: wake up metering agent
1.1.2: create token
1.1.3: meter IoT App & resource usage
1.1.4: write token
1.1.5: send token at U3 intervals
2: release resource & bill
2.1: measure usage quota
2.1.1: request to commit (meter & commit token)
2.1.1.1: publish vote (yes/no)
3: vote from agent A
4: vote from agent B
Commit Request Phase
2PC
Commit Phase
4.1: commit/retry
5: token from agent A
6: token from agent B
7: commit metering transaction
8: summarize & aggregate tokens
9: aggregated usage value
9.1: send for payment
9.1.1: start payment transaction
9.1.2: update client credit

Figure 8.6: DoT Hybrid Metering - 2PC transaction model

In the DoT credit allocation model, the Provisioning server asks the Resource control server to reserve resources and to lock a suitable amount of the user's credit (in case of *pre-paid* model). Then it returns the corresponding amount of credit resources in the form of service specific usage units (e.g., number of invocations, duration) to be metered. The granted and allocated unit type(s) should not be changed during an ongoing DoT session.

8.6.4 Value Verification (VV)

When the end user terminates the service session, the DoT client must send a final service rendering request to the Provisioning server. The Provisioning server should ask the Resource Control server to release all the allocated resources to an IoT application and perform payment transaction. As such, the Resource control server deallocates the granted resources and asks the Metering server to commit the measured metering tokens and report the quota value to it. Then the Resource Control server sends the billable artifacts to the Payment Server to charge the client account respectively. Finally, the Payment server sends the updated client credit to the Resource control server. Meanwhile, the DoT Client drops the user session via AA server.

For the *pay-per-use* model, extra Value Verification interrogations are executed at predefined intervals. This provides a more transparent usage state by frequently reflecting the actual overall usage of services on user account.

In case of *pre-paid* model, upon each deduction from user credit, the DoT protocol verifies the credit value. As soon as the credit value drops below a certain threshold, it informs the user to perform credit update automatically or to take the desired action manually.

8.7 DoT Transaction Model

The runtime of a DoT application is carried out in four "nested chained" transactions. In this respect, the DoT transaction model preserves consistency by defining conflict serializability. In order to prevent the conflict between operations in the DoT transaction model, a schedule pattern is defined to force the logical temporal order in transactions execution. In this case, recoverability is an argument in favor of transaction processing with a rollback mechanism. In the DoT model there exists four transactions prone to inconsistency: (T1) Identification, (T2) Provisioning, (T3) Metering and (T4) Payment transaction. The Identification transaction embeds two sub-transactions of Authorization and Authentication. As such, the Payment transaction has also two sub-transactions of Credit and Debit. Here is the schedule of conflict serializability for read(), write(), and commit() operations on the credit resource object of the client.

We use the following notation: Let T_i be a transaction and *credit* be a relation or a data resource of a relation. Assuming $O_{ij} \in R(credit), W(credit)$ be an atomic read/write operation of T_i on data item *credit*. Then the two operations O_{ij} and O_{ik} on the same *credit* data resource are in conflict if at least one of them is a write operation. To avoid such conflicts, we have come up with the DoT transaction schedule in Table 8.1. One more remark is that we are using the "partial order" function ($\prec$) to specify the execution order between the conflicting operations and between all operations and the termination operation. For instance, the formalization of the DoT metering transaction would be as follows:

T1: Identification	T2: Provisioning	T3: Metering	T4: Payment
:	:	:	:
Lock-S (credit)	Lock-X (credit)	Lock-X (credit)	Lock-X (credit)
Read (credit)	Read (credit)	Read (credit)	Read (credit)
Authenticate & Authorize	Reserve (premium)	Calculate (credit)	Paychain (Credit/Debit)
Unlock (credit)	Deploy (IoT appID)	Write (credit)	Write (credit)
:	Write (credit)	Commit	Commit
	Commit	Unlock (credit)	Unlock (credit)
	Unlock (credit)	:	:
	:		

Table 8.1: The scheduled chronological execution sequence of DoT transactions

$$\sum = Read(credit), Write(credit), Commit$$
$$\prec = (Read(credit), Write(credit)), (Write(credit), Abort), (Read(credit), Abort)$$

$$\sum = Read(credit), Write(credit), Abort$$
$$\prec = (Read(credit), Write(credit)), (Write(credit), Abort), (Read(credit), Abort)$$

8.8 DoT Command Messages

The DoT messages contain commands and Attribute Value Pairs (AVP) to enable metering and payment transactions for DoT-based applications. The messaging structure should be supported by all the collaborating peers in the domain architecture. Four main commands are explained here in more details.

8.8.1 DoT-Request (DoTR) Command

The DoT-Request (DoTR) command is the main command used to send a request among DoT entities. The Request-Type and Requested-Action AVPs are used to determine the purpose of the message and the requested action from the destination DoT entity.

Augmented Backus-Naur Form as defined in RFC5234 (https://tools.ietf.org/html/rfc5234) is used as the metalanguage to define DoT commands, as well as the AVPs which may or must be included. Listing 8.3 specifies the DoT-Request message format.

Listing 8.3: DoT-Request message format in ABNF

```
<DoT-Request> ::== < Diameter Header: DoT Command Code, REQ, PXY >
                   < Session-Id >
                   { Origin-Host }
                   { Origin-Realm }
                   [ Destination-Host ]
                   { Destination-Realm }
                   { Request-Type }
                   { Request-Id }
                   [ Requested-Action ]
                1* [ App-Topology ]
                   [ Subscription-Plan ]
                   [ Accounting-Sub-Session-Id ]
                   [ Acct-Interim-Interval ]
                   [ App-Usage ]
                   [ Confirmation-Mode ]
                   [ Allocation-Unit ]
                   [ User-Name ]
                 * [ Proxy-Info ]
                 * [ Route-Record ]
                 * [ AVP ]
```

8.8.2 DoT-Answer (DoTA) Command

The DoT-Answer (DoTA) command is used between DoT entities to acknowledge the DoTR command. By evaluating the AVPs transferred via this message type, the requestor entity is able to determine whether the requested action was successfully handled or not. Listing 8.4 shows the DoT-Answer message format.

Listing 8.4: DoT-Answer message format in ABNF

```
<DoT-Answer> ::= < Diameter Header: DoT Command Code, PXY >
                 < Session-Id >
                 { Result-Code }
                 { Origin-Host }
                 { Origin-Realm }
                 { Request-Type }
                 { Request-Id }
                 [ User-Name ]
                 [ Error-Message ]
                 [ Error-Reporting-Host ]
                 [ Subscription-Plan ]
                 [ Accounting-Sub-Session-Id ]
                 [ Agent-Vote ]
                 [ App-Usage ]
                 [ User-Action ]
               * [ Failed-AVP ]
               * [ Proxy-Info ]
               * [ AVP ]
```

8.9 Diameter Extended Commands

In addition to the aforementioned DoT commands, we have extended and reused some of the existing commands defined in Diameter base protocol. These commands include AA-Request and Answer, Session-Termination Request and Answer, and Accounting Request and Answer. The following sections briefly describe them.

8.9.1 AA-Request

The AA-Request (AAR) is used by DoT client to request authentication for a given user. The type of request is set to AUTHENTICATE_ONLY (value = 1) using the Auth-Request-Type AVP, since the AA server in DoT protocol is used only for user authentication.

Auth-Session-State MAY be set to NO_STATE_MAINTAINED (value =1). This implies that DoT client SHOULD issue an authentication request each time user submits a new request. Listing 8.5 describes the AA-Request message format.

Listing 8.5: AA-Request message format in ABNF

```
<AA-Request> ::= < Diameter Header: 265, REQ,PXY>
                 < Session-Id >
                 { Auth-Application-Id }
                 { Origin-Host }
                 { Origin-Realm }
                 [ Destination-Host ]
                 { Destination-Realm }
                 { Auth-Request-Type }
                 { User-Name }
                 { Password }
                 [ Authorization-Lifetime ]
                 [ Auth-Grace-Period ]
                 [ Auth-Session-State ]
               * [ Proxy-Info ]
               * [ Route-Record ]
               * [ AVP ]
```

8.9.2 AA-Answer

The AA-Answer (AAA) message is sent in response to the AA-Request (AAR) message. The message format is specified in Listing 8.6

Listing 8.6: AA-Answer message format in ABNF

```
<AA-Answer> ::= < Diameter Header: 265, PXY >
                < Session-Id >
                { Result-Code }
                { Auth-Application-Id }
                { Auth-Request-Type }
                { Origin-Host }
                { Origin-Realm }
                [ User-Name ]
                [ Error-Message ]
                [ Error-Reporting-Host ]
              * [ Failed-AVP ]
                [ Authorization-Lifetime ]
                [ Auth-Grace-Period ]
                [ Auth-Session-State ]
              * [ Proxy-Info ]
              * [ AVP]
```

8.9.3 Session-Termination-Request (STR) Command

The Session-Termination-Request (STR) message is sent by the DoT client to inform the AA Server that an authenticated and/or authorized session is being terminated. This command, as shown in Listing 8.7, is used when the AA server maintains the state.

Listing 8.7: Session-Termination-Request message format in ABNF

```
<ST-Request> ::= < Diameter Header: 275, REQ, PXY >
                 < Session-Id >
                 { Origin-Host }
                 { Origin-Realm }
                 [ Destination-Host ]
                 { Destination-Realm }
                 { Auth-Application-Id }
                 {Termination-Cause }
                 [ User-Name ]
               * [ AVP ]
```

8.9.4 Session-Termination-Answer (STA) Command

The Session-Termination-Answer (STA) message as defined in Listing 8.8, is sent by the AA server to acknowledge that session has been terminated. The Result-Code AVP MUST be present and MAY contain an indication that an error occurred while the STR was being processed. Upon sending or receiving the STA, the Diameter server MUST release all resources for the session indicated by the Session-Id AVP.

Listing 8.8: Session-Termination-Answer message format in ABNF

```
<ST—Answer>  ::= < Diameter Header: 275, PXY >
                 < Session—Id >
                 { Result—Code }
                 { Auth—Application—Id }
                 { Origin—Host }
                 { Origin—Realm }
                 [ User—Name ]
                 [ Error—Message ]
                 [ Error—Reporting—Host ]
               * [ Failed—AVP ]
               * [ AVP ]
```

8.9.5 The Accounting-Request (ACR) Command

The Accounting-Request (ACR) command, as defined in Diameter base protocol, is sent by a Metering agent in order to exchange accounting information with Metering server. The message format is defined as shown in Listing 8.9.

Listing 8.9: Accounting-Request message format in ABNF

```
<AC—Request> ::= < Diameter Header: 271, REQ, PXY >
                 < Session—Id >
                { Origin—Host }
                { Origin—Realm }
                { Destination—Realm }
                { Accounting—Record—Type }
                { Accounting—Record—Number }
                [ Acct—Application—Id ]
                [ Vendor—Specific—Application—Id ]
                [ User—Name ]
                [ Destination—Host ]
                [ Accounting—Sub—Session—Id ]
                [ Acct—Session—Id ]
                [ Acct—Multi—Session—Id ]
                [ Acct—Interim—Interval ]
                [ Accounting—Realtime—Required ]
                [ Origin—State—Id ]
                [ Event—Timestamp ]
              * [ Proxy—Info ]
              * [ Route—Record ]
                { Service—Usage }
                { Resource—Usage}
              * [ AVP ]
```

8.9.6 The Accounting-Answer (ACA) Command

The Accounting-Answer (ACA) command, as defined in Diameter base, is used to acknowledge an Accounting-Request command. Listing 8.10 specifies the message format.

Listing 8.10: Accounting-Answer message format in ABNF

```
<AC-Answer> ::= < Diameter Header: 271, PXY >
                < Session-Id >
                { Result-Code }
                { Origin-Host }
                { Origin-Realm }
                { Accounting-Record-Type }
                { Accounting-Record-Number }
                [ Acct-Application-Id ]
                [ Vendor-Specific-Application-Id ]
                [ User-Name ]
                [ Accounting-Sub-Session-Id ]
                [ Acct-Session-Id ]
                [ Acct-Multi-Session-Id ]
                [ Error-Message ]
                [ Error-Reporting-Host ]
                [ Failed-AVP ]
                [ Acct-Interim-Interval ]
                [ Accounting-Realtime-Required ]
                [ Origin-State-Id ]
                [ Event-Timestamp ]
              * [ Proxy-Info ]
              * [ AVP ]
```

8.10 DoT Attribute-Value Pairs (AVPs)

The following sections describe DoT protocol AVPs, their data formats as defined in Diameter base protocol, and possible values. The AVPs described below are newly defined in DoT, in order to support new functionalities of the protocol. The rest are reused from the Diameter base protocol.

8.10.1 Request-Type AVP

The Request-Type AVP is of type *Enumerated* and indicates the purpose of sending the DoT request message. It must be present in all DoT-Request messages. The following values are defined for this AVP:

- **PROVISION_REQUEST** (value: 1)
  A *provision* request is used to trigger the following actions: to request application topology as well as the subscription plan, to reserve resources upon application submission, and to start IoT application. When the Request-type is set to PROVISION_REQUEST, the Requested_Action AVP must be included in DoTR.

- **METER_REQUEST** (value: 2)
  A *meter* request is used to trigger the following actions: to start IoT application metering, to start metering agents, to commit the measured metering tokens and report the quota value, to retrieve information of granted units as well as the plan

used for initiating the metering process, to terminate IoT application metering, and to commit votes. When the Request-type is set to METER_REQUEST, the Requested_Action AVP MUST be included in DoTR.

- **UPDATE_REQUEST** (value: 3)
  An *update* request is used to trigger the following actions: to confirm or notify the upcoming update request.

- **PAY_REQUEST** (value: 4)
  A *pay* request is used to trigger the following action: to charge the client based on the billable artifacts.

- **TERMINATE_REQUEST** (value: 5)
  A *terminate* request is used to trigger the following actions: to stop IoT application and to release the allocated resources.

8.10.2 Request-Id AVP

The Request-Id AVP is of type Unsigned32 and is used to identify this request within one session. The same as Session-Id AVP, which is globally unique, the combination of Session-Id and Request-Id AVPs must be globally unique. It can be used to match DoT protocol messages with their corresponding acknowledgment (i.e. answers).

8.10.3 Requested-Action AVP

The Requested-Action AVP is of type *Enumerated* and indicates the requested action being sent by DoT-Request command when the Request-Type is METER_REQUEST and PROVISION_REQUEST. The following values are defined for this AVP:

- **PROVISION_APPLICATION_TOPOLOGY** (value: 1)
  This represents a request to provision application topology as well as subscribed plan. It MUST only be used in a DoT message between DoT client and the Provisioning server when the Request-Type AVP is set to PROVISION_REQUEST.

- **APP_RESOURCE_ALLOCATION** (value: 2)
  This represents a request to reserve resources including underlying resources as well as credit. This is for the prepaid model. It MUST only be used in a DoT message between Provisioning server and the Resource control server with a Request-Type AVP set to PROVISION_REQUEST.

- **START_IOT_APP** (value: 3)
  This presents a request to start IoT application. It MUST only be used in a DoT message between DoT client and the Provisioning server with a Request-Type AVP set to PROVISION_REQUEST.

- **START_APP_METERING** (value: 4)
  This presents a request to start IoT application metering. It MUST only be used in a DoT message between the Provisioning server and the Metering server with a Request-Type AVP set to METER_REQUEST.
- **GET_APP_SPECIFICATION** (value: 5)
  This presents a request to retrieve the information of subscribed plan including granted units as well as U3 rate. It MUST only be used in a DoT message between Metering server and the Resource control server with a Request-Type AVP set to METER_REQUEST.
- **START_METERING_AGENT** (value: 6)
  This presents a request to start the Metering agent as well as metering process. It MUST only be used in a DoT message between Metering server and Metering Agents with a Request-Type AVP set to METER_REQUEST.
- **PUBLISH_USAGE_UPDATE** (value: 7)
  This presents a request to publish the latest usage update. It MUST only be used in a DoT message between Metering server and the Resource control server with a Request-Type AVP set to METER_REQUEST.
- **SEND_COMMIT_VOTES** (value: 8)
  This presents a request to implement the first phase of 2PC. It MUST only be used in a DoT message between Metering server and the Metering agents with a Request-Type AVP set to METER_REQUEST.
- **COMMIT_METERING_TOKEN** (value: 9)
  This presents a request to implement the second phase of 2PC. It MUST only be used in a DoT message between Metering server and the Metering agents with a Request-Type AVP set to METER_REQUEST.
- **COMMIT_APP_METERING** (value: 10)
  This presents a request to commit the measured metering tokens and report the quota value. It MUST only be used between a Resource control server and the Metering server with a Request-Type AVP set to METER_REQUEST.
- **TERMINATE_APP_METERING** (value: 11)
  This presents a request to terminate IoT application metering. It MUST only be used between Provisioning server and the Metering server with a Request-Type AVP set to METER_REQUEST.

8.10.4 App-Topology AVP

The App-Topology AVP is of type OctetString and contains the IoT Application Topology submitted to the DoT client. It is in the form of a URI pointing to the repository that contains the topology files. The topology is originally defined using JSON-LD format.

8.10.5 Subscription-Plan AVP

The Subscription-Plan AVP is of type OctetString and contains the subscription plan submitted to the DoT client. It is in the form of a URI pointing to the repository that contains the subscription plan file. The plan is originally defined using JSON-LD format.

8.10.6 App-Usage AVP

The App-Usage AVP is of type Float64 and contains the application usage in the form of credit usage.

8.10.7 Agent-Vote AVP

The Agent-Vote AVP is of type Enumerated.

- **NO** (value: 0)
  The agent is not ready to submit the token (e.g. agent experiences a failure that will make it impossible to commit)
- **YES** (value: 1)
  The agent is ready to submit the token (commit)

8.10.8 Confirmation-Mode AVP

The Confirmation-Mode AVP is of type *Enumerated.* It MUST be provided in an update request messages so that DoT servers can decide if they should wait for the user-action or if this is just a notification. The following values are supported:

- **NOTIFY_ONLY** (value: 0)
  The update request will proceed automatically and the user needs to be only notified. No user action is expected.
- **USER_ACTION_REQUIRED** (value: 1)
  The update procedure needs user confirmation to proceed.

8.10.9 Allocation-Unit AVP

The Allocation-Unit AVP is of type *Enumerated.* It may be provided in an update request messages to notify the receiver about the type of a resource unit that needs to be updated. The ultimate receiver of the update message, here the DoT client, can use this value to send messages to end user by explicitly specifying the type of unit which needs to be updated.

- **CREDIT** (value: 0)
  The allocation unit is for prepaid models.

- **SUBSCRIPTION_TIME** (value: 1)
  The allocation unit is for pay-per-use models.

8.10.10 User-Action AVP

The User-Action AVP is of type *Enumerated.* It MUST be provided in responses to the request messages that the Confirmation-Mode is set to USER_ACTION_REQUIRED.

- **REJECT** (value: 0)
  The end user rejects the request.
- **CONFIRM** (value: 1)
  The end user confirms the request.

8.10.11 Password AVP

The Password AVP is of type OctetString and contains the password of the user to be authenticated.

As required in Diameter base protocol, Diameter messages are encrypted using IPsec or TLS. Message encryption is important for Password AVP, as it contains sensitive information.

8.10.12 Service-Usage AVP

The Service-Usage AVP is of type *Unsigned64* and contains the service usage value in the duration or the number of invocation, for instance.

8.10.13 Resource-Usage AVP

The Resource-Usage AVP is of type *Grouped* AVP and has the following structure as shown in Listing 8.11:

Listing 8.11: Resource-Usage grouped AVP format in ABNF

```
Resource-Usage::= < AVP Header: DoT AVP Code >
                  [ CPU-Usage ]
                  [ Memory-Usage ]
                  [ Network-Received ]
                  [ Network-Sent ]
                  [ Filesystem-Usage ]
                  [ Uptime ]
                * [ AVP ]
```

8.10.14 CPU-Usage AVP

The CPU-Usage AVP is of type *Unsigned64* and contains the cumulative CPU usage of all cores in nanoseconds.

8.10.15 Memory-Usage AVP

The Memory-Usage AVP is of type *Unsigned64* and contains the total memory usage in bytes.

8.10.16 Network-Received AVP

The Network-Received AVP is of type *Unsigned64* and contains the total number of bytes received over the network.

8.10.17 Network-Sent AVP

The Network-Sent AVP is of type *Unsigned64* and contains the total number of bytes sent over the network.

8.10.18 Filesystem-Usage AVP

The Filesystem-Usage AVP is of type *Unsigned64* and contains the total number of bytes used on file-system.

8.10.19 Uptime AVP

The Uptime AVP is of type Unsigned64 and contains the number of milliseconds since the agent's container has been started.

8.11 DoT Mandatory AVPs

DoT commands define variety of AVPs which are mandatory for some requests and not required for others. This section describes the mandatory AVPs and their appropriate values based on the request type for DoT Request and DoT Answer messages.

8.11.1 Provisioning Requests

A provisioning request includes the following variations:

A. Provisioning Application Topology
The *Provisioning application topology* request is sent by the DoT client to the Provisioning server in order to provision application topology as well as subscribed plan.

The DoT client MUST set the Request-Type AVP to PROVISION_REQUEST, and Requested-Action to PROVISION_APPLICATION_TOPOLOGY. In addition, it MUST include App-Topology and Subscription-Plan AVPs in DoT-Request message.

The Provisioning server receiving this request sends a DoT-Answer message back to acknowledge the successful provision of the IoT Application. No extra AVPs need to be included.

B. Allocating Resources to the Application
The *Allocating resources to the application* request is sent by the Provisioning server to the Resource control server to reserve resources including underlying resources as well as credit in case of prepaid model.

The Provisioning server MUST set the Request-Type AVP equal to PROVISION_REQUEST, and Requested-Action to APP_RESOURCE_ALLOCATION. In addition, it MUST include App-Topology and Subscription-Plan AVPs in DoT-Request message.

The Resource control server receiving this request sends a DoT-Answer back to acknowledge the successful allocation of required resources. No extra AVPs need to be included.

C. Starting IoT Application
The *Starting IoT Application* request is sent by the DoT client to the Provisioning server to start IoT application. The DoT client MUST set the Request-Type AVP equal to PROVISION_REQUEST, and Requested-Action to START_IOT_APP. No extra AVPs need to be included.

The Provisioning server receives this request and sends a DoT-Answer back to confirm the running status of running services. No extra AVPs need to be included.

8.11.2 Metering Requests

The Metering requests can be of the following types:

A. Start Application Metering
The *Start application metering* request is sent by the Provisioning server to the Metering server to start IoT application metering. The Provisioning server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to START_APP_METERING. No extra AVPs need to be included.

The Metering server receives this request and sends a DoT-Answer back as an acknowledge to confirm the application is being metered. No extra AVPs need to be included.

B. Get Application Specification
The *Get application specification* request is sent by the Metering server to the Resource control server to retrieve the information of subscribed plan including granted units as well as U3 rate of each microservice.

The Metering server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to GET_APP_SPECIFICATION. No extra AVPs need to be included.

The Resource control server receives this request and sends a DoT-Answer back and includes the Plan.Therefore the Subscription-Plan AVP MUST be included in the answer message.

C. Start Metering Agent
The *Start metering agent* request is sent by the Metering server to the Metering agents in order to request to start the Metering agent as well as metering process.

The Metering server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to START_METERING_AGENT. In addition, it MUST include Accounting-Sub-Session-Id and Acct-Interim-Interval AVPs in DoT-Request message. Acct-Interim-Interval AVP which is defined in Diameter base protocol SHALL be used to send U3 rate to the agent.

The Metering agents receives this request and sends a DoT-Answer back as an acknowledge to confirm the service is being metered .In addition, it MUST include Accounting-Sub-Session-Id.

D. Publish Usage Update
The *Publish usage update* request is sent by the Metering server to the Resource control server to publish the latest usage update.

The Metering server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to PUBLISH_USAGE_UPDATE. In addition, it MUST include App-Usage AVP in DoT-Request message.

The Resource control server receives this request and sends a DoT-Answer back to acknowledge the receipt of usage data. No extra AVPs need to be included.

E. Send Commit Votes
The *Send commit votes* request is sent by the Metering server to the Metering agents in order to implement the first phase of 2PC, in which the coordinator (i.e., Metering server) sends a "*query to commit*" message to all cohorts (i.e., Metering agents) and waits until it receives a reply from all cohorts.

The Metering server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to SEND_COMMIT_VOTES. In addition, it MUST include Accounting-Sub-Session-Id AVP

The Metering agents receives this request and sends a DoT-Answer back as an answer containing the agent vote. Therefore, it MUST include Agent-Vote AVP, as well as the Accounting-Sub-Session-Id AVP.

F. Commit Metering Token
The *Commit metering token* request is sent by the Metering server to the Metering agents

in order to implement the second phase of 2PC, in which the coordinator (i.e. Metering server) sends a "*commit*" message to all cohorts (i.e. Metering agents).

The Metering server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to COMMIT_METERING_TOKEN. In addition, it MUST include Accounting-Sub-Session-Id AVP.

The Metering agents receives this request and sends a DoT-Answer back as an answer to acknowledge the receipt of COMMIT_METERING_TOKEN request. It MUST include Accounting-Sub-Session-Id AVP.

G. Commit App Metering
The *Commit app metering* request is sent by the Resource control server to the Metering server to commit the measured metering tokens and report the quota value.

The Resource control server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to COMMIT_APP_METERING. No extra AVPs need to be included.

The Metering server receives this request and sends a DoT-Answer back as an answer including the quota value. It MUST therefore include App-Usage AVP.

H. Terminate App Metering
The *Terminate app metering* request is sent by the Provisioning server to the Metering server to terminate IoT application metering.

The Provisioning server MUST set the Request-Type AVP equal to METER_REQUEST, and Requested-Action to TERMINATE_APP_METERING. No extra AVPs need to be included.

The Metering server receives this request and sends a DoT-Answer back as an acknowledge confirming termination of metering. No extra AVPs need to be included.

8.11.3 Update Request

The *Update request* is sent from the Resource control server to the Provisioning server, and from the Provisioning server to the DoT client in order to request an upcoming update in credit amount or subscription time via Allocation-Unit AVP. The Resource control server constantly checks the current running state against certain thresholds. To be more specific, in pre-paid model, the available credit amount is checked with a certain threshold. However, for the pay-per-use model, the subscription time is checked. The threshold value is predefined in provider's policy configuration file and can be overwritten by the subscription plan. When the aforementioned threshold is reached, DoT protocol sends a DoT request to notify user if an automatic update is enabled by user in subscription plan. Otherwise, the DoT protocol asks the user to take the considerable action.

When Confirmation-Mode is set to USER_ACTION_REQUIRED, the requestor servers wait for the response message which contains the User-Action. During this time if the unit, which the update request has been sent for, is fully consumed and no response indicating user action is received, DoT protocol MUST terminate the service. These will stop any further updates.

The Resource control server and the Provisioning server, when acting as a requestor, MUST set the Request-Type AVP equal to UPDATE_REQUEST. They MUST also include the Confirmation-Mode AVP. They MAY include Allocation-Unit.

The DoT client and the Provisioning server, when acting as the receiver of the update request, send a DoT-Answer back as an acknowledge. The DoT-Answer is sent immediately when the Confirmation-Mode AVP in request is set to NOTIFY_ONLY, otherwise it waits for the user action. In the latter case the User-Action AVP MUST be included in the answer message.

8.11.4 Payment Request

Payment request is sent by the Resource control server to the Payment server to charge the client based on the generated billable artifacts. The Resource control server MUST set the Request-Type AVP equal to PAY_REQUEST. It MUST also include the App-Usage AVP.

The Payment server receives this request and sends a DoT-Answer back as an acknowledge confirming the state of the payment transaction. No extra AVPs need to be included.

8.11.5 Terminate Request

Terminate Request is sent from the DoT client to the Provisioning server, and from Provisioning server to the Resource control server to release allocated resources and stop IoT application.

The DoT client and the Provisioning server, when acting as a requestor, MUST set the Request-Type AVP equal to TERMINATE_REQUEST. No extra AVPs need to be included.

The Resource control server and the Provisioning server, when acting as the receiver of the terminate request, send a DoT-Answer back to acknowledge the receipt of terminate request. No extra AVPs need to be included.

8.12 DoT State Machines

This section defines the DoT application protocol state machines. There are five different state machines for each entity in DoT protocol. The first state machine as shown in Table 8.3 describes the states of DoT client. The second one describes the Provisioning server state machine. The third one is the state machine of Resource control server. The fourth and fifth state machines describe the Metering and Payment servers accordingly.

The events of *Tw expired*, *Failure to send* and *temporary error* in the state machines indicate the situation where there is a problem in network, preventing one entity to communicate with the desired one. It also might be the cases where the destination entity is too busy and cannot handle the request at that time.

Furthermore, for the sake of brevity the following abbreviations are used instead of full name of the request types. Table 8.2 summarizes the abbreviations.

Abbr.	Full Name
PAT	PROVISION_APPLICATION_TOPOLOGY
ARA	APP_RESOURCE_ALLOCATION
SIA	START_IOT_APP
CAM	COMMIT_APP_METERING
SAM	START_APP_METERING
GAS	GET_APP_SPECIFICATION
PUU	PUBLISH_USAGE_UPDATE
TAM	TERMINATE_APP_METERING
SMA	START_METERING_AGENT
SCV	SEND_COMMIT_VOTES
CMT	COMMIT_METERING_TOKEN
UR	UPDATE_REQUEST
TR	TERMINATE_REQUEST
PR	PAY_REQUEST

Table 8.2: The abbreviations used in state machines

The DoT client state machine is shown in Table 8.3. The states PendingI, PendingP, PendingD, PendingT correspond to the pending states to wait for an answer to an already sent request related to *Identification*, *Provisioning*, *Deployment*, *Termination* requests.

State	Event	Action	New State
Idle	AA request received from end user	Send AA request to AA server, start Tw	PendingI
pendingI	Successful AA answer received	Submit application topology as well as the plan to Provisioning sever in DoT-Request (Requested-Action: PAT), restart Tw	PendingP
pendingI	Tw expired, Failure to send, temporary error	Retry sending AA request to AA server, restart Tw	PendingI
PendingP	DoT-Answer received from Provisioning server (Requested-Action: PAT)	Submit DoT-Request to start the IoT application to Provisioning server (Requested-Action: SIA),restart Tw	PendingD
PendingP	DoT-Answer received from Provisioning server (Requested-Action: PAT) with Result-Code != SUCCESS	Fix the problem and send the DoT-Request again, restart Tw	PendingD
PendingD	DoT-Answer received from Provisioning server (Requested-Action: SIA)	Stop Tw, inform end user that service, monitoring has started	Open
PendingD	DoT-Answer received from Provisioning server (Requested-Action: SIA) with Result-Code != SUCCESS	Fix the problem and send the DoT-Request again, restart Tw	PendingD
PendingD	Tw expired, Failure to send, temporary error	Retry sending DoT-Request to Provisioning server (Requested-Action: SIA), restart Tw	pendingD
Open	DoT-Request received from Provisioning server (Request-Type: UR) with Confirmation-Mode = NOTIFY_ONLY	Inform user about the update, send DoT-Answer back to Provisioning server (Request-Type: UR)	Open
Open	DoT-Request received from Provisioning server (Request-Type: UR) with Confirmation-Mode = USER_ACTION_REQUIRED	Send update confirmation request to user	Open
Open	DoT-Request received from Provisioning server (Request-Type: UR) but not successfully processed	Send DoT-Answer back to Provisioning server (Request-Type: UR) with Result-Code != SUCCESS	Open
Open	User confirmed the update	Send update confirmation answer (Request-Type: UR) with User-Action = CONFIRM to Provisioning server	Open
Open	User rejected the update	Send update confirmation answer (Request-Type: UR) with User-Action = REJECT to Provisioning server	Open
Open	User sends termination request	Send termination request to Provisioning server (Request-Type: TR), start Tw	PendingT

continues on next page ...

continued from previous page ...

State	Event	Action	New State
PendingT	DoT-Answer received from Provisioning server (Request-Type: TR)	Inform user about termination, stop Tw	Idle
PendingT	Answer received from Provisioning server (Request-Type: TR) with Result-Code != SUCCESS	Fix the problem and send the request again, restart Tw	PendingT
PendingT	Tw expired, Failure to send, temporary error	Retry sending termination request to Provisioning server (Request-Type: TR), restart Tw	PendingT

Table 8.3: DoT Client state machine

Next, the Provisioning server state machine is described in Table 8.4. The states of PendingR and PendingM correspond to the states of waiting for an answer from the Resource control server and the Metering server respectively.

State	Event	Action	New State
Idle	DoT-Request to provision application topology from DoT client (Requested-Action: PAT) is received and successfully processed	Send the resource allocation DoT-Request to Resource control server (Requested-Action: ARA), Start Tw	PendingR
Idle	DoT-Request to provision application topology from DoT client (Requested-Action: PAT) is received but not successfully processed	Send the DoT-Answer to DoT client (Requested-Action: PAT) with Result-Code != SUCCESS	Idle
PendingR	DoT-Answer of resource allocation from Resource control server (Requested-Action: ARA) is received	Send the DoT-Answer to DoT client (Requested-Action: PAT), Stop Tw	Idle
PendingR	DoT-Answer of resource allocation from Resource control server (Requested-Action: ARA) is received with Result-Code != SUCCESS	Fix the problem and send the resource allocation DoT-Request to Resource control server (Requested-Action: ARA) again, Restart Tw	PendingR
PendingR	Tw expired, Failure to send, temporary error	Restart Tw, Retry sending the resource allocation DoT-Request to Resource control server (Requested-Action: ARA)	PendingR
Idle	DoT-Request to start the IoT application from DoT client (Requested-Action: SIA) is received and successfully processed	Send the start metering DoT-Request to Metering server (Requested-Action: SAM) , Start Tw	PendingM
Idle	DoT-Request to start the IoT application from DoT client (Requested-Action: SIA) is received but not successfully processed	Send theDoT-Answer to DoT client (Requested-Action: SIA) with Result-Code != SUCCESS	Idle
PendingM	DoT-Answer of starting IoT application from Metering server (Requested-Action: SAM) is received	Send the DoT-Answer to DoT client (Requested-Action: SIA), Stop Tw	Open
PendingM	DoT-Answer of starting IoT application from Metering server (Requested-Action: SAM) is received with Result-Code != SUCCESS	Fix the problem and send the start metering DoT-Request to Metering server (Requested-Action: SAM) again, Restart Tw	PendingM

continues on next page ...

continued from previous page ...

State	Event	Action	New State
PendingM	Tw expired, Failure to send, temporary error	Restart Tw, Retry sending the start metering DoT-Request to Metering server (Requested-Action: SAM) again	PendingM
Open	DoT-Request to notify user about updating resource allocation from Resource control server (Request-Type: UR) is received and successfully processed	Send the DoT-Request as allocation notification to DoT client (Request-Type: UR)	Open
Open	DoT-Request to terminate IoT application from DoT client (Request-Type: TR) is received and successfully processed	Send the resource release DoT-Request to Resource control server (Request-Type: TR), Start Tw	PendingR
Open	DoT-Request to terminate IoT application from DoT client (Request-Type: TR) is received but not successfully processed	Send the terminate app DoT-Answer to DoT client (Request-Type: TR) with Result-Code != SUCCESS	Open
PendingR	DoT-Answer of confirming the release of allocated resources from Resource control server (Request-Type: TR) is received	Send the terminate metering DoT-Request to Metering server (Requested-Action: TAM), Start Tw	PendingM
PendingR	DoT-Answer of confirming the release of allocated resources from Resource control server (Request-Type: TR) is received with Result-Code != SUCCESS	Fix the problem and send the resource release DoT-Request to Resource control server (Request-Type: TR) again, Restart Tw	PendingR
PendingR	Tw expired, Failure to send, temporary error	Restart Tw, Retry sending DoT-Request to release the allocated resources to Resource control server (Request-Type: TR)	PendingR
PendingM	DoT-Answer of confirming metering termination from Metering server (Requested-Action: TAM) is received	Send the terminate app DoT-Answer to DoT client (Request-Type: TR), Stop Tw	Idle
PendingM	DoT-Answer of confirming metering termination from Metering server (Requested-Action: TAM) is received with Result-Code != SUCCESS	Fix the problem and send the terminate metering DoT-Request to Metering server (Requested-Action: TAM) again, Restart Tw	PendingM
PendingM	Tw expired, Failure to send, temporary error	Terminate the service, Send the terminate app DoT-Answer to DoT client (Request-Type: TR)	Idle

Table 8.4: Provisioning server state machine

The Resource control server state machine is shown in table 8.5. The states of PendingPY and PendingPM correspond to the state of waiting for the payment and metering answer.

State	Event	Action	New State
Idle	DoT-Request to reserve resources from Provisioning server (Requested-Action: ARA) is received and successfully processed	Perform resource scheduling, reserve resources, lock the credit, send the resources allocation as DoT-Answer to Provisioning server (Requested-Action: ARA)	Open
Idle	DoT-Request to reserve resources from Provisioning server (Requested-Action: ARA) is received but not successfully processed	send the resources allocation DoT-Answer to Provisioning server (Requested-Action: ARA) with Result-Code != SUCCESS	Idle
Open	DoT-Request to retrieve the Specification information from Metering server (Requested-Action: GAS) is received and successfully processed	Send the DoT-Answer containing granted units and plan to Metering server (Requested-Action: GAS)	Open
Open	DoT-Request to retrieve the Specification information from Metering server (Requested-Action: GAS) is received but not successfully processed	Send the getting specification DoT-Answer to Metering server (Requested-Action: GAS) with Result-Code != SUCCESS	Open
Open	DoTRequest to publish usage update from Metering server (Requested-Action: PUU) is received and successfully processed	Charge credit according the last sent usage, Send the acknowledge DoT-Answer to Metering server (Requested-Action: PUU)	Open
Open	DoT-Request to publish usage update from Metering server (Requested-Action: PUU) is received but not successfully processed	Send the usage update DoT-Answer to Metering server (Requested-Action: PUU) with Result-Code != SUCCESS	Open
Open	Credit threshold is met (in pre-paid model)	Update and lock credit, Send update notification in a DoT-Request to Provisioning server (Request-Type: UR)	Open
Open	Subscription time threshold is met (in pay-per-use model)	Send update notification in a DoT-Request to Provisioning server (Request-Type: UR)	Open
Open	DoT-Request to release the allocated resources from Provisioning server (Request-Type: TR) is received and successfully processed	Release the allocated resources, Send the commit request in a DoT-Request to Metering server (Requested-Action: CAM), Start Tw	PendingM
Open	DoT-Request to release the allocated resources from Provisioning server (Request-Type: TR) is received but not successfully processed	Send a DoT-Answer to Provisioning server (Request-Type: TR) with Result-Code != SUCCESS	Open
PendingM	DoT-Answer of commiting metering containing the quota values from Metering server (Requested-Action: CAM) is received	Send a DoT-Answer Provisioning server (Requested-Action: TR), Send a DoT-Request to Payment server to charge the client based on billable artifact (Request-Type: PR), Restart Tw	PendingPY

continues on next page ...

continued from previous page ...

State	Event	Action	New State
PendingM	DoT-Answer of commiting metering from Metering server (Requested-Action: CAM) is received with Result-Code !=SUCCESS	Fix the problem and send aDoT-Request to Metering server (Requested-Action: CAM) again, Restart Tw	PendingM
PendingM	Tw expired, Failure to send, temporary error	Retry sending the DoT-Request to Metering server (Requested-Action: CAM) , Restart Tw	PendingM
PendingPY	DoT-Answer of billing payment from Payment server (Request-Type: PR) is received	StopTw	Idle
PendingPY	DoT-Answer of billing payment from Payment server (Request-Type: PR) is received with Result-Code != SUCCESS	Fix the problem and send a DoT-Request to Payment server (Request-Type: PR) again, Restart Tw	PendingPY
PendingPY	Tw expired, Failure to send, temporary error	Retry sending aDoT-Request to Payment server (Request-Type: PR), Restart Tw	PendingPY

Table 8.5: Resource control server state machine

The Metering server state machine is presented in Table 8.6. The states PendingG and PendingUU stand for pending states for Granted unit and Usage Update information

State	Event	Action	New State
Idle	DoT-Request to start metering received from Provisioning server (Requested-Action: SAM)	Send DoT-Answer to the Provisioning server (Requested-Action: SAM), send DoT-Request to Resource control server asking for granted Units and plan (Requested-Action: GAS), start Tw	PendingG
Idle	DoT-Request to start metering received (Requested-Action: SAM) but not successfully processed	Send DoT-Answer to the Provisioning server (Requested-Action: SAM) with Result-Code != SUCCESS	Idle
PendingG	DoT-Answer received from Resource control server including granted units and plan (Requested-Action: GAS)	Send service specific U3 to each metering agent, start metering transaction, stop Tw	Open
PendingG	DoT-Answer received from Resource control server (Requested-Action: GAS) with Result-Code !=SUCCESS	Fix the problem and send the request again, restart Tw	PendingG
PendingG	Tw expired, Failure to send, temporary error	Retry sending DoT-Request (Requested-Action: GAS) again, restart Tw	PendingG
Open	Unit usage update message received from a Metering agent	Summarize the usage and send it in a DoT-Request to Resource control server (Requested-Action: PUU), start Tw	PendingUU
PendingUU	DoT-Answer received from Resource control server (Requested-Action: PUU)	Stop Tw	Open

continues on next page ...

continued from previous page ...

State	Event	Action	New State
PendingUU	Tw expired, Failure to send, temporary error	Retry sending DoT-Request (Requested-Action: PUU) again, restart Tw	PendingUU
PendingUU	DoT-Answer received from Resource control server (Requested-Action: PUU) with Result-Code !=SUCCESS	Fix the problem and send the DoT-Request again (Requested-Action: PUU), restart Tw	PendingUU
Open	DoT-Request to commit the measured metering tokens and report the quota value received from Resource control server (Requested-Action: CAM)	Aggregate tokens and send the DoT-Answer back to Resource control server (Requested-Action: CAM)	Open
Open	DoT-Request received (Requested-Action: CAM) but not successfully processed	Send the DoT-Answer back to Resource control server (Requested-Action: CAM) with Result-Code != SUCCESS	Open
Open	DoT-Request to terminate metering received from Provisioning server (Request-Type:TR)	Terminate metering, send DoT-Answer back to Provisioning server (Request-Type: TR)	Idle

Table 8.6: Metering server state machine

State	Event	Action	New State
Idle	DoT-Request to charge the client based on billable artifact from Resource control server (Request-Type: PR) is received and successfully processed	Generate a bill by Invoking the payment transaction, Deduct credit from the end user's account and refund unused reserved credit to the user's account, Send DoT-Answer to Resource control server (Request-Type: PR)	Idle
Idle	DoT-Request to charge the client based on billable artifact from Resource control server (Request-Type: PR) is received but not successfully processed	Send DoT-Answer to Resource control server (Request-Type: PR) with Result-Code != SUCCESS	Idle

Table 8.7: Payment server state machine

8.13 Related Work

In relation to our work, there is some commendable research regarding the cloud service usage metering. Elmsroth et al.[78] proposed a loosely coupled architecture solution for an accounting and billing system for use in the RESERVOIR[79] project. There are some alternatives that propose billing and metering solutions, Narayan et al.[81]. Petersson[82] describes cloud metering and billing solution. Naik et al.[83] proposed a solution for metering of services delivered from multiple cloud providers. They incorporate the cloud service broker together with a metering control system to report metered data at configurable intervals. Meanwhile, there are some prominent studies on capturing pricing models for IoT domains. Aazam et el.[92] provided a resource prediction, resource price estimation and reservation for the Fog which resides between underlying IoTs and the cloud. Their proposed model does not support the real-time session and event-based metering models.

Mazhelis et al.[93] studies the applicability of the Constrained Application Protocol (CoAP), a lightweight transfer protocol under development by IETF, for efficiently retrieving monitoring and accounting data from constrained devices. Their results indicate that CoAP is suited for efficiently transferring monitoring and accounting data, both due to a small energy footprint and a memory-wise compact implementation. This work relies on using the accounting and monitoring infrastructure produced by the Accounting and Monitoring of Authentication and Authorization Infrastructure Services (AMAAIS) project[94]. Our work elevates the metering to an IoT domain by proposing an extended Diameter protocol that enables IoT infrastructures to incorporate the DoT protocol in their deployment models. This contributes to a real-time resource utilization awareness by constructing and allocating flexible metering models to IoT deployment units.

8.14 Conclusion

In this study, we have presented a metering protocol, called the Diameter of Things (DoT), that enables real-time telemetry and automatic resource allocation control of IoT applications. The DoT is designed to enhance the resource utilization by extending the Diameter base protocol. The authors offered the DoT architecture, its interrogations as well as its transaction model for accounting for the resource usage of constrained devices. The DoT offers considerable benefits, such as granular metering of lightweight applications, real-time transparency over resource usage in edge devices and transporting and exchanging application-specific metering policies in an IoT domain. As an outlook, our future work includes the DoT implementation together with an evaluation of the proposed DoT architecture in terms of scalability, performance and session management. A concise evaluation of the requirements[5] for the Diameter-based protocols is proposed that will be considered for the DoT implementation as well as evaluation purposes. We envision the DoT protocol to gaining acceptance as a de facto metering standard in IoT.

[5]https://tools.ietf.org/html/draft-zander-ipfix-diameter-eval-00

CHAPTER 9

Conclusions

Here, we conclude this thesis, noting the principles, mechanisms and models applied in our contributions. We then present our decisions and findings to demonstrate how the state of the art in research has been advanced as part of this thesis work. The work presented in this thesis constitutes three steps in bridging the gap between Cyber-physical Systems (CSP) and business value propositions with the goal of continuous Edge to Business (E2B) value chain delivery.

9.1 Review of Contributions

The overall aim of this thesis is to investigate the feasibility of exposing CPS applications as metered services to the clients via cloud-enabled business plans.

9.1.1 Part I: IoT Busines Models

In chapter 3, we consider data as a core business model design material and explore opportunities on how to make them tradable entities. We elaborate the government data and expose them as services, using a platform called GoDaaS. We define seven types of data-driven business models of GoDaaS, namely, *Data Infrastructure as a Service (IaaS)*, *Storage as a Service (STaaS)*, *Data as a Service (DaaS)*, *Database as a Service (DBaaS)*, *Platform as a Service (PaaS)*, *Search as a Service (SEaaS)* and *Data Analytics as a Service (DAaaS)*. All seven business models and their associated elements are consistently specified and modeled in details, mostly from the developer's view. Government data is usually unstructured and available in diversity of formats. As a reaction to this complexity, we designed a new abstraction layer called Data Compute Unit (DCU), that allows governments to express their data packages more structured and consistent, so that developers can utilize these packages by treating them as objects. This ultimately eases the data resource sharing and trading. GoDaaS multi-layer architecture needs an integration enabler for its implementation and a uniform service delivery model. Government service bus (GSB) layer glues all the entities, agencies and services together through its messaging and queuing mechanisms.

In chapter 4, we focus on defining a proper marketable entity which covers all aspects of a product. Cloud manufacturing provides a cooperative work environment for enterprises and individuals, enabling collaboration among the entire manufacturing ecosystem. With the cloud, resource pools can use virtualization to abstract away the heterogeneousness and regional distribution of manufacturing resources. Cloud manufacturing aims to orchestrate and allocate such distributed resources and render production services for clients to seamlessly enable manufacturing on demand.

Manufacturing resources (such as devices, sensors, materials, and drivers) are virtualized and encapsulated into a product bill of materials (BOM). Manufacturers can access, configure, invoke, deploy, and orchestrate this BOM on distributed production lines in a near real-time manner. BOM data must go to the right manufacturer at the right time for the right cost.

Our proposed mechanism deals with the provisioning, portability, and management of all types of manufacturing resources as services, for all phases of the production lifecycle. This mechanism incorporates the Oasis Topology and Orchestration Specification for Cloud Applications (TOSCA) policies, plans, and templates as a mechanism for dynamic configuration, portability, and management of product BOMs across multiple collaborating manufacturers. Once virtualized on the cloud, we refer to a product BOM as a bill of manufacturing services (BOMS).

In chapter 5, we enrich the sensor raw data to become more interpretable. We have developed a middleware, called Gatica that collects the real-time sensor data via gateways, enriches them using annotations then transforms and exposes them in RDF triples. The linked data RDFs are then streamed to the analytics endpoints for querying and reasoning purposes. We have applied this model in the healthcare system. Having such data in place, the health-care service providers are able to construct a wellness-function for the normal range of the vitals and produce alerts upon on deviating from the normal values. Through this vital range interpretation, various disease patterns can be discovered together with its severity.

9.1.2 Part II: IoT Design Patterns

In chapter 6, we propose eight design patterns with an edge-based focus in mind. Such patterns advices CPS application engineers to with some design best practices on how to build their IoT ecosystem. This basically brings a unified design principles for common class of IoT systems' architectural challenges. The patterns are defined in a unified pattern language conventions.

The IoT patterns address various aspects of design. For instance, we propose the container-based virtualization method as an optimal method for edge services provisioning. They basically contain the baseline environment to run the service together with all dependencies and configurations. We then define an edge footprint messaging pattern for a resource-constrained environment. Such messaging mechanisms benefit from reducing the message header size and response codes, minimizing the set of methods for exchanging data.

Next, we focus on an in-device data storage and processing patterns. The CPS has a long data-processing pipeline in terms of collection, storage, and processing, and the decisions made at the earlier stages of the pipeline can significantly impact the processing at later stages. For the lightweight data processing, we are proposing a Pipe and Filter architecture. There are three filters in lightweight data processing: data points validation, cleansing, and aggregation. In Pipe and Filter architecture, the output of one filter is fed, as input, into the next filter. In this pipeline the first filter does data validation. The task is done by submitting a validation job to a spark master. The validation job is done by spark workers. The next filter is cleansing. Based on the different cleaning policies, missing and invalid values may be deleted or replaced with a reasonable value. Data aggregation is the next. The output of these phases must satisfy the quality of data constraints.

9.1.3 Part III: Micro Telemetry

In chapter 7, we leverage the micro telemetry mechanism to data-intensive applications. The level of a resource metering unit becomes an essential facet of facilitating a way of hierarchical metering and processing of usage patterns indexed by information granules. By granule metering, we mean a collection of metrics aggregated together by their functional relationship or closeness. Such granules are then formulated by adopting and leveraging a certain level of abstraction to achieve further utility. Each abstraction level is formed by grouping metrics together into semantically meaningful constructs to reflect the structure of the original data into its granular counterpart. Granular metering enables diving deeper into measuring the resource usage on the per-DAG-flow, per-DAG, per-Job, per-Map or per-Reduce levels. Such metering granules can be regarded as more abstract and interpretable entities in charging clients and in elasticity policy enforcement. We treat them as a scale unit. This provides users real-time visibility over their resource consumption and the ongoing money stream being paid as they go. Furthermore, it enables clients realtime application control to ensure that quality and cost constraints are met.

We then develop an elastic data telemetry system that enables granular metering and automatic control of MapReduce applications due to their current behavior and preset configurations. TED is designed to enhance the resource utilization using YARN cluster queueing system. It observes the running job's progress, interprets its cost and quality behavior, audits the predefined job requirements. Then it generates a tailored resource allocation policy with regard to capacity constraints and moves the job to the potential queues to scale in or up.

In chapter 8, we address forward IoT resource planning which is an evolving facet of utility computing that aims to leverage and treat computing resources as metered services. Furthermore, metering is the process of measuring and recording the utilization rate of an entire application, individual parts of an application, or tasks and underlying resources. Along with this idea, respecting resource capacity constraints on edge devices establishes a firm requirement for a protocol in support of a telemetry of IoT applications. Such metering capability is needed when lack of resources among competing applications dictates our schedule and credit allocation.

In this chapter, the authors offer the Diameter of Things (DoT) protocol that can be incorporated to implement a real-time metering of IoT services in the case of prepaid as well as pay-per-use economic models. The protocol employs a mechanism to handle time-based and event-based telemetry patterns. The former is used for scenarios where the charged units are continuously consumed while the latter is typically used when units are implicit invocation events. The DoT-enabled platform performs a chained metering transaction on a graph of dependent IoT microservices, collects the emitted usage data, then generates billable artifacts. Finally it permits micropayments to take place in parallel.

9.2 Future Research Directions

In this thesis work, various solutions for enabling the Edge to Business (E2B) value chain have been proposed, namely; IoT business models, diversity of design artifacts as architectural patterns for building CPS applications, and micro telemetry solutions to capture the values generated fro edge infrastructures. Yet, a number of open issue remain which were out of scope of this thesis. As an outlook, in the following we conclude the thesis by summarizing such challenges for future research focus.

Extending the work presented in this thesis, one may extend our business models to support more domain applications as well as creating novel business opportunities from the CPS context data. Another extension may include the role of human-based service in this E2B value chain.

Moving forward, there are many opportunities on defining novel design patterns for the resource-constrained environments. Such design artifacts may be classified as elasticity, scalability, availability, resiliency, software defined networking and security patterns for edge applications.

On the micro edge telemetry research, there are huge opportunities to extend the work. For instance, one can focus on how to meter the edge device usage. For instance, measuring the battery usage since many embedded devices and connected sensors run on batteries and need to conserve electricity. Such metering model has a direct influence on resource-constrained applications elasticity.

Bibliography

[1] J. Rivera and R. van der Meulen, “In 2020, 25 Billion Connected ’Things’ Will Be in Use.” http://www.gartner.com/newsroom/id/2905717, 2014, [Online; accessed 2-November-2015].

[2] C. P. S. P. W. Group, “DRAFT Framework for Cyber-Physical Systems.” http://www.nist.gov/el/nist-releases-draft-framework-cyber-physical-systems-developers.cfm, 2015, [Online; accessed 2-November-2015].

[3] T. G. P. M. Mell;, vol. The NIST Definition of Cloud Computing. NIST SP - 800-145, September 28, 2011.

[4] T. Jones, “Virtualization for embedded system,” http://www.ibm.com/developerworks/library/l-embedded-virtualization/, 2011, [Online; accessed 2-November-2015].

[5] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC ’12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[6] Y. C. Lee, Y. Kim, H. Han, and S. Kang, “Fine-grained, adaptive resource sharing for real pay-per-use pricing in clouds,” in *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, Sept 2015, pp. 236–243.

[7] E. Bucherer and D. Uckelmann, “10 Business Models for the Internet of Things,” *Business*, pp. 1–25, 2011.

[8] C. Bormann and M. Ersue, “Terminology for Constrained-Node Networks,” https://tools.ietf.org/html/rfc7228, May 2014, [Online; accessed 2-November-2015].

[9] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, “Portable cloud services using tosca,” *Internet Computing, IEEE*, vol. 16, no. 3, pp. 80–85, May 2012.

[10] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, “Cloud computing: Distributed internet computing for it and scientific research,” *Internet Computing, IEEE*, vol. 13, no. 5, pp. 10–13, 2009.

[11] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[12] P. Mell and T. Grance, "The nist definition of cloud computing (draft)," *NIST Special Publication*, vol. 800, p. 145, 2011.

[13] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli, "Modeling cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 13 – 28, January 2012.

[14] C. P. S. P. W. Group, "DRAFT Framework for Cyber-Physical Systems," http://www.nist.gov/el/nist-releases-draft-framework-cyber-physical-systems-developers.cfm, 2015, [Online; accessed 19-October-2015].

[15] M. Satyanarayanan, "Pervasive computing: vision and challenges," *Personal Communications, IEEE*, vol. 8, no. 4, pp. 10–17, Aug 2001.

[16] "Osterwalder, alexander; pigneur, yves; and tucci, christopher l. "business model generation," alexander osterwalder, yves pigneur, alan smith, and 470 practitioners from 45 countries, self published, 2010."

[17] D. Wu, D. W. Rosen, L. Wang, and D. Schaefer, "Cloud-based design and manufacturing," *Comput. Aided Des.*, vol. 59, no. C, pp. 1–14, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.cad.2014.07.006

[18] X. Xu, "From cloud computing to cloud manufacturing," *Robotics and Computer-Integrated Manufacturing*, vol. 28, no. 1, pp. 75 – 86, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0736584511000949

[19] P. Lipton and S. Moser, "OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC," https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca, Last accessed: 2015.08.04.

[20] M. Burgess, *In Search of Certainty - The Science of Our Information Infrastructure*, 1st ed. XtAxis Press, 2013.

[21] E. Commission, "Study on business models for linked open government data - BM4LOGD," https://joinup.ec.europa.eu/node/72473, 12 November 2013, [Online; accessed 24 June 2014].

[22] N. Shadbolt and K. O'Hara, "Linked data in government," *Internet Computing, IEEE*, vol. 17, no. 4, pp. 72–77, July 2013.

[23] M. Vafopoulos and M. Meimaris, "Weaving the economic linked open data," in *Semantic and Social Media Adaptation and Personalization (SMAP), 2012 Seventh International Workshop on*, Dec 2012, pp. 92–97.

[24] J. Hoxha and A. Brahaj, “Open government data on the web: A semantic approach,” in *Emerging Intelligent Data and Web Technologies (EIDWT), 2011 International Conference on*, Sept 2011, pp. 107–113.

[25] E. Kalampokis, E. Tambouris, and K. Tarabanis, “On publishing linked open government data,” in *Proceedings of the 17th Panhellenic Conference on Informatics*, ser. PCI ’13. New York, NY, USA: ACM, 2013, pp. 25–32. [Online]. Available: http://doi.acm.org/10.1145/2491845.2491869

[26] M. Vafopoulos, “A framework for linked data business models,” in *Informatics (PCI), 2011 15th Panhellenic Conference on*, Sept 2011, pp. 95–99.

[27] D. DiFranzo, A. Graves, J. Erickson, L. Ding, J. Michaelis, T. Lebo, E. Patton, G. Williams, X. Li, J. Zheng, J. Flores, D. McGuinness, and J. Hendler, “The web is my back-end: Creating mashups with linked open government data,” in *Linking Government Data*, D. Wood, Ed. Springer New York, 2011, pp. 205–219. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-1767-5_10

[28] F. Leymann, “Linked compute units and linked experiments: Using topology and orchestration technology for flexible support of scientific applications,” in *Software Service and Application Engineering*, ser. Lecture Notes in Computer Science, M. Heisel, Ed. Springer Berlin Heidelberg, 2012, vol. 7365, pp. 71–80. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30835-2_6

[29] B. Nicolae, P. Riteau, and K. Keahey, “Bursting the cloud data bubble: Towards transparent storage elasticity in iaas clouds,” in *28th IEEE International Parallel & Distributed Processing Symposium*, Phoenix, AZ, 2014.

[30] A. Fuggetta, G. P. Picco, and G. Vigna, “Understanding code mobility,” *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[31] A. Thomas, D. Trentesaux, and P. Valckenaers, “Intelligent distributed production control,” *Journal of Intelligent Manufacturing*, vol. 23, no. 6, pp. 2507–2512, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10845-011-0601-x

[32] L. Zhang, Y. Luo, F. Tao, B. H. Li, L. Ren, X. Zhang, H. Guo, Y. Cheng, A. Hu, and Y. Liu, “Cloud manufacturing: a new manufacturing paradigm,” *Enterprise Information Systems*, vol. 8, no. 2, pp. 167–187, 2014.

[33] C. Brecher, W. Lohse, and M. Vitr, “Module-based platform for seamless interoperable cad-cam-cnc planning,” in *Advanced Design and Manufacturing Based on STEP*, ser. Springer Series in Advanced Manufacturing, X. Xu and A. Y. C. Nee, Eds. Springer London, 2009, pp. 439–462. [Online]. Available: http://dx.doi.org/10.1007/978-1-84882-739-4_20

[34] Q. Li, C. Wang, J. Wu, J. Li, and Z.-Y. Wang, “Towards the business-information technology alignment in cloud computing environment: Anapproach based on collaboration points and agents,” *Int. J. Comput. Integr. Manuf.*, vol. 24, no. 11, pp. 1038–1057, Nov. 2011.

[35] S. Schulte, D. Schuller, R. Steinmetz, and S. Abels, “Plug-and-Play Virtual Factories,” *IEEE Internet Computing Magazine*, vol. 16, no. 5, pp. 78–82, 2012. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MIC.2012.114

[36] S. Schulte, P. Hoenisch, C. Hochreiner, S. Dustdar, and M. Klusch, “Towards Process Support for Cloud Manufacturing,” in *18th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2014) Ulm, Germany.* IEEE Computer Society, Washington, DC, USA, 2014, pp. NN–NN.

[37] X. Wang and X. Xu, “Dimp: An interoperable solution for software integration and product data exchange,” *Enterp. Inf. Syst.*, vol. 6, no. 3, pp. 291–314, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1080/17517575.2011.587544

[38] ——, “Icms: A cloud-based manufacturing system,” in *Cloud Manufacturing*, ser. Springer Series in Advanced Manufacturing, W. Li and J. Mehnen, Eds. Springer London, 2013, pp. 1–22. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-4935-4_1

[39] D. Sow, D. Turaga, and M. Schmidt, “Mining of sensor data in healthcare: A survey,” in *Managing and Mining Sensor Data*, C. C. Aggarwal, Ed. Springer US, 2013, pp. 459–504. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-6309-2_14

[40] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, “Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals,” *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000. [Online]. Available: http://circ.ahajournals.org/content/101/23/e215.abstract

[41] B. D. Abdi, Herve, *Principal Component and Correspondence Analyses Using R*, ser. SpringerBriefs in Statistics, 2015. [Online]. Available: http://www.springer.com/gp/book/9783319092553

[42] R. Watanabe, E. Morett, and E. Vallejo, “Inferring modules of functionally interacting proteins using the bond energy algorithm,” *BMC Bioinformatics*, vol. 9, no. 1, 2008. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-9-285

[43] S. Banerjee, A. Mishra, and R. Dasgupta, “Semantic exploration of sensor data,” in *Proceedings of the 5th International Workshop on Web-scale Knowledge Representation Retrieval &#38; Reasoning*, ser. Web-KR ’14. New York, NY, USA: ACM, 2014, pp. 55–58. [Online]. Available: http://doi.acm.org/10.1145/2663792.2663800

[44] D. Le-Phuoc, J. Xavier Parreira, and M. Hauswirth, “Linked stream data processing,” in *Reasoning Web. Semantic Technologies for Advanced Query Answering*, ser. Lecture Notes in Computer Science, T. Eiter and T. Krennwallner, Eds. Springer Berlin Heidelberg, 2012, vol. 7487, pp. 245–289. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33158-9_7

[45] D. Le-Phuoc, M. Dao-Tran, M. . Pham, P. Boncz, T. Eiter, and M. Fink, *Linked stream data processing engines: Facts and figures*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2012, vol. 7650 LNCS, no. PART 2. [Online]. Available: www.scopus.com

[46] P. Barnaghi, W. Wang, L. Dong, and C. Wang, “A linked-data model for semantic sensor streams,” in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, Aug 2013, pp. 468–475.

[47] D. Le-Phuoc, H. Q. Nguyen-Mau, J. X. Parreira, and M. Hauswirth, “A middleware framework for scalable management of linked streams,” *Web Semant.*, vol. 16, pp. 42–51, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.websem.2012.06.003

[48] Fielden, *G.D.R, Engineering Design*, ser. London: HMSO, 1975.

[49] D. Bandyopadhyay and J. Sen, “Internet of things: Applications and challenges in technology and standardization,” *Wireless Personal Communications*, vol. 58, no. 1, pp. 49–69, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11277-011-0288-5

[50] T. Xu, J. B. Wendt, and M. Potkonjak, “Security of iot systems: Design challenges and opportunities,” in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 417–423. [Online]. Available: http://dl.acm.org/citation.cfm?id=2691365.2691450

[51] D. Fuller, “System design challenges for next generation wireless and embedded systems,” in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE ’14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 1:1–1:1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616608

[52] S. C. Mukhopadhyay, *Internet of Things: Challenges and Opportunities.* Springer Publishing Company, Incorporated, 2014.

[53] H. Lamaazi, N. Benamar, A. Jara, L. Ladid, and D. El Ouadghiri, “Challenges of the internet of things: Ipv6 and network management,” in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference on*, July 2014, pp. 328–333.

[54] R. S. Hanmer, “Pattern mining patterns,” in *Proceedings of the 19th Conference on Pattern Languages of Programs*, ser. PLoP ’12. USA: The Hillside Group, 2012, pp. 23:1–23:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821679.2831293

[55] T. Wellhausen and A. Fiesser, “How to write a pattern?: A rough guide for first-time pattern authors,” in *Proceedings of the 16th European Conference on Pattern Languages of Programs*, ser. EuroPLoP ’11. New York, NY, USA: ACM, 2012, pp. 5:1–5:9. [Online]. Available: http://doi.acm.org/10.1145/2396716.2396721

[56] G. Meszaros and J. Doble, “Pattern languages of program design 3,” R. C. Martin, D. Riehle, and F. Buschmann, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ch. A Pattern Language for Pattern Writing, pp. 529–574. [Online]. Available: http://dl.acm.org/citation.cfm?id=273448.273487

[57] B. Di Martino, “Applications portability and services interoperability among multiple clouds,” *Cloud Computing, IEEE*, vol. 1, no. 1, pp. 74–77, May 2014.

[58] R. Lea and M. Blackstock, “City hub: A cloud-based iot platform for smart cities,” in *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, Dec 2014, pp. 799–804.

[59] S. Qanbari, S. Mahdizadeh, R. Rahimzadeh, N. Behinaein, and S. Dustdar, “Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications,” http://www.gecon-conference.org/gecon2015/images/papers/qanbari_paper_25.pdf, 2015, [Online; accessed 19-October-2015].

[60] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns.* New York, NY, USA: John Wiley & Sons, Inc., 1996.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[62] R. Hanmer, *Patterns for Fault Tolerant Software.* Wiley Publishing, 2007.

[63] V.-P. Eloranta and M. Leppänen, “Patterns for distributed machine control systems,” in *Proceedings of the 18th European Conference on Pattern Languages of Program*, ser. EuroPLoP ’13. New York, NY, USA: ACM, 2015, pp. 6:1–6:15. [Online]. Available: http://doi.acm.org/10.1145/2739011.2739017

[64] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer Publishing Company, Incorporated, 2014.

[65] K. Hashizume, N. Yoshioka, and E. B. Fernandez, “Misuse patterns for cloud computing,” in *Proceedings of the 2Nd Asian Conference on Pattern Languages of*

Programs, ser. AsianPLoP '11. New York, NY, USA: ACM, 2011, pp. 12:1–12:6. [Online]. Available: http://doi.acm.org/10.1145/2524629.2524644

[66] K. Hashizume, E. B. Fernandez, M. M. Larrondo-Petrie, and E. B. Fernandez, "Cloud service model patterns," in *Proceedings of the 19th Conference on Pattern Languages of Programs*, ser. PLoP '12. USA: The Hillside Group, 2012, pp. 10:1–10:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821679.2831280

[67] K. Tamagawa, "AWS cloud design patterns," http://en.clouddesignpattern.org, 2008, [Online; accessed 19-October-2015].

[68] L. B. M. N. T. S. Alex Homer, John Sharp, "Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications," https://msdn.microsoft.com/en-us/library/dn568099.aspx, 2014, [Online; accessed 19-October-2015].

[69] B. Wilder, "Cloud Architecture Patterns," http://oreil.ly/cloud_architecture_patterns, 2012, [Online; accessed 19-October-2015].

[70] "Cloud computing concepts, technology and architecture by thomas erl, zaigham mahmood and ricardo puttini," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 4, pp. 37–38, Aug. 2014, reviewer-Gvero, Igor. [Online]. Available: http://doi.acm.org/10.1145/2632434.2632462

[71] A. N. R. P. Thomas Erl, Zaigham Mahmood, "Cloud Computing Concepts, Technology and Architecture," http://cloudpatterns.org, 2013, [Online; accessed 19-October-2015].

[72] M. Koster, "Design Patterns for an Internet of Things," http://community.arm.com/groups/internet-of-things/blog/2014/05/27/design-patterns-for-an-internet-of-things, 2014, [Online; accessed 19-October-2015].

[73] M. Rappa, "The utility business model and the future of computing services," *IBM Systems Journal*, vol. 43, no. 1, pp. 32–42, 2004.

[74] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.future.2008.12.001

[75] S. Sen, C. Joe-Wong, S. Ha, and M. Chiang, "A survey of smart data pricing: Past proposals, current plans, and future trends," *ACM Comput. Surv.*, vol. 46, no. 2, pp. 15:1–15:37, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2543581.2543582

[76] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[77] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, “Principles of elastic processes,” *IEEE Internet Computing*, vol. 15, no. 5, pp. 66–71, 2011.

[78] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera, “Accounting and billing for federated cloud infrastructures,” in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, ser. GCC ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 268–275. [Online]. Available: http://dx.doi.org/10.1109/GCC.2009.37

[79] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán, “The reservoir model and architecture for open federated cloud computing,” *IBM J. Res. Dev.*, vol. 53, no. 4, pp. 535–545, Jul. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1850659.1850663

[80] S. Prasad and S. Avinash, “Smart meter data analytics using opentsdb and hadoop,” in *Innovative Smart Grid Technologies - Asia (ISGT Asia), 2013 IEEE*, Nov 2013, pp. 1–6.

[81] A. Narayan, S. Rao, G. Ranjan, and K. Dheenadayalan, “Smart metering of cloud services,” in *Systems Conference (SysCon), 2012 IEEE International*, March 2012, pp. 1–7.

[82] J. Petersson, “Cloud metering and billing,” http://www.ibm.com/developerworks/cloud/library/cl-cloudmetering [Online; accessed 08-August-2011].

[83] V. Naik, K. Beaty, and A. Kundu, “Service usage metering in hybrid cloud environments,” in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, pp. 253–260.

[84] “Calhoun, p., johansson, t., perkins, c., hiller, t.,and p. mccann, "diameter mobile ipv4 application", rfc 4004, august 2005.” *IETF.*

[85] “Hakala, h., mattila, l., koskinen, j-p., stura, m., and j. loughney, "diameter credit-control application", rfc 4006, august 2005.” *IETF.*

[86] “Garcia-martin, m., "diameter session initiation protocol (sip) application", rfc 4740, april 2006.” *IETF.*

[87] “C. rigney, a. rubens, w. simpson, s. willens, "remote authentication dial in user service (radius)", rfc2138, june 2000.” *IETF.*

[88] “Calhoun, p., loughney, j., guttman, e., zorn, g., and j. arkko, "diameter base protocol", rfc 3588, september 2003.” *IETF.*

[89] R. L. Rivest and A. Shamir, “Payword and micromint: Two simple micropayment schemes,” in *Proceedings of the International Workshop on Security Protocols*. London, UK, UK: Springer-Verlag, 1997, pp. 69–87. [Online]. Available: http://dl.acm.org/citation.cfm?id=647214.720369

[90] M. Voegler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "DIANE - dynamic iot application deployment," in *2015 IEEE International Conference on Mobile Services, MS 2015, New York City, NY, USA, June 27 - July 2, 2015*, 2015, pp. 298–305. [Online]. Available: http://dx.doi.org/10.1109/MobServ.2015.49

[91] C. Inzinger, S. Nastic, S. Sehic, M. Voegler, F. Li, and S. Dustdar, "Madcat: A methodology for architecture and deployment of cloud application topologies," in *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, April 2014, pp. 13–22.

[92] M. Aazam and E.-N. Huh, "Fog computing micro datacenter based dynamic resource estimation and pricing model for iot," in *Advanced Information Networking and Applications (AINA), 2015 IEEE 29th International Conference on*, March 2015, pp. 687–694.

[93] O. Mazhelis, M. Waldburger, G. S. Machado, B. Stiller, and P. Tyrväinen, "Retrieving monitoring and accounting information from constrained devices in internet-of-things applications," in *Proceedings of the 7th IFIP WG 6.6 International Conference on Autonomous Infrastructure, Management, and Security: Emerging Management Mechanisms for the Future Internet - Volume 7943*, ser. AIMS'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 136–147. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38998-6_17

[94] "Stiller, b.: Accounting and monitoring of aai services. switch journal 2010(2) (october 2010) 12-13," *IETF*.

Curriculum Vitae

Mr. Soheil Qanbari

Research Assistant/ PhD Candidate. at Distributed Systems Group

Technical University of Vienna (TU Wien)

soheil@dsg.tuwien.ac.at

[Schwemmgasse 2/3/55A - 1020 Wien]

[Mobile: +43 (650) 480 3584]

Senior Research/Management Associate Jobs

- **Cyber-Physical Systems (CPS)**
  Edge Engineering, IoT Design Patterns, Embedded Systems, Smart City, Cloud Manufacturing.
- **Cloud Computing**
  Cloud Service Models, Elastic computing, Transaction Processing, Cloud Patterns, Service Metering.
- **Big Data Analytics**
  Hadoop, YARN, Spark, Stream Processing, MapReduce, NoSQL, In-Device Data Processing.

Academic Education

- **Technical University of Vienna (TU Wien)** Vienna, Austria. ID: 1329639
  ***PhD.** in Cloud & Edge Computing ; GPA:1.54 Winter. 2012 – Planned Defense, January 2016*
  - **Key Research Focus**: Distributed Systems Technologies, IoT PaaS for Smart Cities, Cloud Resource Elasticity, Embedded Virtualization, Portable Cloud Manufacturing Services, Government Data as a Service, Cloud Business Models, Cloud Asset Pricing Models.
  - **PhD Dissertation**: Title: *Edge-to-Business Value Chain Delivery via Elastic Telemetry of Cyber-Physical Systems*. Thesis will be furnished upon request.
  - **Key Courses**: Advanced Internet Computing, Business Intelligence, Recommender Systems, Advanced Services Engineering, Interdisciplinary Research Seminar.
- **Baha'i Institute for Higher Education (BIHE)** Tehran, Iran. ID: 7722221789
  ***MSc.** in Software Engineering.; GPA:3.58 on 4.0 scale. Autumn. 2007 – Spring. 2010*
  - **Master Thesis**: Verification & Validation on Meta Models using Model-driven Ontology Development. Thesis will be furnished upon request.
  - **Key Courses**: Decision Support System, Software Project Management, Robotics, Data Mining, Advanced Requirement Engineering, Semantic Web, Information System Security, Research Methodology, Master's Thesis
- **Baha'i Institute for Higher Education (BIHE)** Tehran, Iran. ID: 7721031789
  ***BSc.** in Computer Engineering.; GPA:2.1 on 3.0 scale. Autumn. 1998 – Autumn. 2004*
  - **Key Courses**: Artificial Intelligence, Internet Engineering, Database Design & Management, Real time Systems, Software Engineering, System Analysis & Design, Compiler Design, Object Oriented Programming, Design of Algorithms, Data Structures & Algorithms, Calculus, Advanced Engineering Mathematics.

Course Teaching & Lecturing Experience

- **Big Data Computing** BIHE University
  Course Instructor Winter Semester 2014
  - Data-intensive Computing with Hadoop & YARN Ecosystem..
  - NoSQL, NewSQL, Cassandra, Hbase, Hadoop HDFS
  - MapReduce Programming Model, R & Pig Latin.
  - Oozie Workflow Scheduler & Zookeeper Distributed Coordination Service
  - Big Data Search with Lucene, SolrCloud and ElasticSearch
  - Big Data Integration with Pentaho Kettle, Sqoop, Flume.
  - Predictive Analysis with Apache Mahout.
  - Stream Computing with Spark.

Course Teaching & Lecturing Experience (cont'd)

- **Advanced Software Architecture** — BIHE University
  Course Instructor — *Summer Semester 2015*
  - Software Architecture Styles.
  - Message Oriented Middle-ware Architecture.
  - Transaction Processing Models.
  - Quality of Service (QoS): Scalability, Availability & Consistency.
  - Software Architecture Patterns.
  - Cloud Computing Design Patterns.
  - Internet of Things (IoT) Application Architecture.
  - Service Oriented Architecture (SOA).
  - Microservice Architecture.
- **Enterprise Architecture** — BIHE University
  Course Instructor — *Summer Semester 2010*
  - Zachman & TOGAF Enterprise-Architecture Methodologies.
  - Enterprise Resource Planning (ERP) Solution Modeling.
  - Process Hierarchy Diagram (PhD).
- **Object Oriented Programming** — BIHE University
  Course Instructor — *Winter Semester 2008*
  - GoF Design Patterns.
  - Enterprise Integration Patterns.

Professional Experience

- **TU Wien Distributed Systems Group/Smart City Project** — Vienna, Austria
  Research Assistant at Pacific Controls Cloud Computing Lab — *July. 2012 – till Dec. 2015*
  - Smart City PaaS Development using WSO2.
  - Hybrid IoT Compute Unit (Cloud Services, Edge Services, Human Services).
  - Smart City Business Messaging using WSO2 FIX Gateway and ESB.
  - Software-defined Internet of Things (SD-IoT).
  - IoT Design Patterns to Design, Build and Engineer Edge Applications.
  - Telemetry of IoT Services using Diameter of Things (DoT) Protocol.
  - Linked Sensor Data Streaming/Processing.
- **BIHE University** — Tehran, Iran
  IT Office Manager — *Aug. 2008 – May. 2011*
  - University wide eLearning Infrastructure Management.
  - Learning Management Systems (LMS) Management.
- **Haseb System Engineering Consultancy Incorporation** — Tehran, Iran
  Enterprise Solution Architect — *Oct. 2006 – Jun. 2008*
  - Enterprise architecture modeling using Zachman framework
  - Enterprise Architecture Blueprint and ERP Solution Gap Analysis/Mapping
  - Integrator of Pentaho business intelligent suite and ERP solutions
  - Intalio BPMS Designer
- **Data Processing Douran Company** — Tehran, Iran
  Project Manager/ ERP Implementor — *Oct. 2002 – Aug. 2006*
  - ERP Technical Solution Architect
  - Enterprise Business Process Designer
  - Compiere/Adempiere ERP Suite Localization
  - Color Manufacturing ERP Implementation
  - Car Manufacturing ERP Implementation

Frameworks, Skills

Languages: C/C++, J2EE, Python, OWL, NodeJS.

Methodologies: IBM DAD, RUP, Agile, Scrum.

Modeling Tools: Enterprise Architect, Visual Paradigm, MySQL Workbench, MagicDraw (UML).

Databases: RedisDB, BerkleyDB, MySQL, Oracle Fusion Middleware, PostgreSQL, MongoDB, OpenTSDB, HBase, Casandra, Hive.

Team working: Git, Bitbucket, Jira, IssueTracker, SVN, Microsoft Project.

Standards/Technology: BPEL, BPMN, RDF, TOSCA, UML, XML, JSON, CBOR, CoAP, Web services (SOAP, RESTful), CEP, Semantic web, SOA.

Integration: Message Oriented Middleware (MOM), Queue-based Messaging, JMS, Topic, CoAP, Machine 2 Machine (M2M), ActiveMQ, MQTT, OpenDDS, Pentaho Kettle. Apache Sqoop.

Development Environment: Eclipse, PyCharm, WebStorm, ProtegeOWL.

Frameworks: OpenStack, WSO2, OpenHAB, Spring MVC, Twisted, SedonaDev, Cloudera Hadoop, YARN, Apache Tez, WEKA, Apache ActiveMQ.

Research Publications

- Contribution to Standards/Protocols:
  - **Soheil Qanbari**, Samira Mahdizadeh, Negar Behinaein, Rabee Rahimzadeh and Schahram Dustdar, **Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications**, Internet Engineering Task Force (IETF). Draft submission!

- Conference Proceedings:
  2015
  - **Soheil Qanbari**, Samim Pezeshki, Rozita Raisi, Samira Mahdizadeh, Rabee Rahim zadeh, Negar Behinaein, Fada Mahmoudi, Shiva Ayoubzadeh, Parham Fazlali, Keyvan Roshani, Azalia Yaghini, Mozhdeh Amiri, Ashkan Farivarmoheb, Arash Zamani, and Schahram Dustdar, **IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications**, *IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI 2016)*, April 4-8, Berlin, Germany. In review.
  - **Soheil Qanbari**, Samira Mahdizadeh, Rabee Rahimzadeh, Negar Behinaein, and Schahram Dustdar, **Diameter of Things (DoT): A Protocol for Real-time Telemetry of IoT Applications**, 12th International Conference on Economics of Grids, Clouds, Systems, and Service (GECON 2015), (GECON-Conf, Cluj-Napoca, Romania, 15-17 September.
  - **Soheil Qanbari**, Ashkan Farivarmoheb, Parham Fazlali, Samira Mahdizadeh and Schahram Dustdar, **Telemetry for Elastic Data (TED): Middleware for MapReduce Job Metering and Rating**, The 9th IEEE International Conference on Big Data Science and Engineering (BigDataSE) (IEEE BigDataSE 2015), Helsinki, Finland, 20-22 August, 2015.

Research Publications (cont'd)

- **Soheil Qanbari**, Negar Behinaein, Rabee Rahimzadeh and Schahram Dustdar, **Gatica: Linked Sensed Data Enrichment and Analytics Middleware for IoT Gateways**, IEEE International Conference on Future Internet of Things and Cloud (IEEE FiCloud 2015), August 24-26, 2015, Rome, Italy.
- **Soheil Qanbari**, Navid Rekabsaz and Schahram Dustdar, **Open Government Data as a Service (GoDaaS): Big Data Platform for Mobile App Developers**, IEEE International Conference on Open and Big Data (IEEE OBD 2015), August 24-26, 2015, Rome, Italy.

2014

- **Soheil Qanbari**, Samira Mahdi Zadeh, Soroush Vedaei and Schahram Dustdar. **CloudMan: A Platform for Portable Cloud Manufacturing Services**, 2014 IEEE International Conference on BigData (IEEE BigData 2014), 27-30 Oct, 2014, Washington DC, United States.
- **Soheil Qanbari**, Vahid Sebtoo, Schahram Dustdar. **Cloud Resources-Events-Agents Model: Towards TOSCA-based Applications**, Third European Conference on Service-Oriented and Cloud Computing (ESOCC 2014), 2-4 Sept, 2014, Manchester, United Kingdom.
- **Soheil Qanbari**, Fei Li, Schahram Dustdar, Tian-Shyr Dai. **Cloud Asset Pricing Tree (CAPT): Elastic Economic Model for Cloud Service Providers**, 4th International Conference on Cloud Computing and Services Science (CLOSER 2014), 3-5 April, 2014, Barcelona, Spain. **Best student paper award!**
- Dustdar S., Voegler M., Sehic S., **Qanbari S.**, Nastic S., Truong H.-L. (2014). **The Internet of Things Meets Cloud Computing in Smart Cities**. Bridges, Vol. 41, (invited).

2013

- Li F., Vgler M., Sehic S., **Qanbari S.**, Truong H.-L., Nastic S., Dustdar S. (2013). **IoT PaaS: Intelligent IT infrastructure for smart cities**. in: Smart City - Viennese Expertise based on Science and Research, Schmid (publisher), Vienna, (invited), ISBN: 978-3-900607-51-7.
- Dustdar S., Li F., Truong H.-L., Sehic S., Nastic S., **Qanbari S.**, Voegler M., Claessens M. (2013). **Green Software Services: From Requirements to Business Models** (keynote with invited paper). 2nd International Workshop on Green and Sustainable Software (GREENS 2013) in conjunction with ICSE 2013, May 18-26, 2013 San Francisco, CA, USA.

• Journal Articles:

- **Soheil Qanbari**, Fei Li, and Schahram Dustdar. **Toward Portable Cloud Manufacturing Services**, Internet Computing, IEEE, vol. 18, no. 6, pp.77-80, 2014.
- Fei Li, Michael Voegler, Sanjin Sehic, **Soheil Qanbari**, Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar, **Web-Scale Service Delivery for Smart Cities**, Internet Computing, IEEE, vol. 17, no. 4, pp.78-83, 2013.

• Book Chapters:

- Fei Li, **Soheil Qanbari**, Michael Voegler and Schahram Dustdar, "**Constructing green software services: From service models to cloud-based architecture**," in Green in Software Engineering, (invited) : Springer International Publishing Switzerland, 2014.

Research Publications (cont'd)

– Fei Li, Michael Voegler, Sanjin Sehic, **Soheil Qanbari**, Hong-Linh Truong, Stefan Nastic, Schahram Dustdar, "**IoT PaaS: Intelligente IT-Infrastruktur fuer Smart Cities**", in: "smart city - Wiener Know-How aus Wissenschaft und Forschung", Schmid Verlag, Wien, 2012, (invited), ISBN: 978-3-900607-50-0, 191-198.

Language Proficiency

Persian: Mother tongue; **English:** Level C2; **German:** Level B1; **Arabic:** Level A2