NNT : 2016 EMSE 0813

THÈSE

présentée par

Andrei CIORTEA

pour obtenir le grade de
Docteur de l'École Nationale Supérieure des Mines de Saint-Étienne
en cotutelle avec Université « Politehnica » de Bucarest

Spécialité : Informatique

TISSER LE WEB SOCIAL DES OBJETS : PERMETTRE UNE INTERACTION AUTONOME ET FLEXIBLE DANS L'INTERNET DES OBJETS

soutenue à Bucarest, le 14 janvier 2016

Membres du jury

Président :	Emil SLUȘANSCHI	Professeur, Université Politehnica, Bucarest
Rapporteurs :	Fabien GANDON	DR HDR, INRIA, Sophia Antipolis
	Erik MANNENS	Professeur, Université de Gand, Gand
Examinateur(s) :	Alessandro RICCI	Professeur Associé, Université de Bologne, Bologne
	Costin BĂDICĂ	Professeur, Université de Craiova, Craiova
	Ștefan TRĂUȘAN-MATU	Professeur, Université Politehnica, Bucarest
Directeur(s) de thèse :	Adina Magda FLOREA	Professeur, Université Politehnica, Bucarest
	Olivier BOISSIER	Professeur, École Nationale Supérieure des Mines, Saint-Etienne
	Antoine ZIMMERMANN	Maître Assistant, Ecole Nationale Supériere des Mines, Saint-Etienne

Spécialités doctorales	**Responsables :**	**Spécialités doctorales**	**Responsables**
SCIENCES ET GENIE DES MATERIAUX	K. Wolski Directeur de recherche	MATHEMATIQUES APPLIQUEES	O. Roustant, Maître-assistant
MECANIQUE ET INGENIERIE	S. Drapier, professeur	INFORMATIQUE	O. Boissier, Professeur
GENIE DES PROCEDES	F. Gruy, Maître de recherche	IMAGE, VISION, SIGNAL	JC. Pinoli, Professeur
SCIENCES DE LA TERRE	B. Guy, Directeur de recherche	GENIE INDUSTRIEL	A. Dolgui, Professeur
SCIENCES ET GENIE DE L'ENVIRONNEMENT	D. Graillot, Directeur de recherche	MICROELECTRONIQUE	S. Dauzere Peres, Professeur

EMSE : Enseignants-chercheurs et chercheurs autorisés à diriger des thèses de doctorat (titulaires d'un doctorat d'État ou d'une HDR)

ABSI	Nabil	CR	Génie industriel	CMP
AVRIL	Stéphane	PR2	Mécanique et ingénierie	CIS
BALBO	Flavien	PR2	Informatique	FAYOL
BASSEREAU	Jean-François	PR	Sciences et génie des matériaux	SMS
BATTON-HUBERT	Mireille	PR2	Sciences et génie de l'environnement	FAYOL
BERGER DOUCE	Sandrine	PR2	Sciences de gestion	FAYOL
BERNACHE-ASSOLLANT	Didier	PR0	Génie des Procédés	CIS
BIGOT	Jean Pierre	MR(DR2)	Génie des Procédés	SPIN
BILAL	Essaid	DR	Sciences de la Terre	SPIN
BLAYAC	Sylvain	MA(MDC)	Microélectronique	CMP
BOISSIER	Olivier	PR1	Informatique	FAYOL
BORBELY	Andras	MR(DR2)	Sciences et génie des matériaux	SMS
BOUCHER	Xavier	PR2	Génie Industriel	FAYOL
BRODHAG	Christian	DR	Sciences et génie de l'environnement	FAYOL
BRUCHON	Julien	MA(MDC)	Mécanique et ingénierie	SMS
BURLAT	Patrick	PR1	Génie Industriel	FAYOL
COURNIL	Michel	PR0	Génie des Procédés	DIR
DARRIEULAT	Michel	IGM	Sciences et génie des matériaux	SMS
DAUZERE-PERES	Stéphane	PR1	Génie Industriel	CMP
DEBAYLE	Johan	CR	Image Vision Signal	CIS
DELAFOSSE	David	PR0	Sciences et génie des matériaux	SMS
DESRAYAUD	Christophe	PR1	Mécanique et ingénierie	SMS
DOLGUI	Alexandre	PR0	Génie Industriel	FAYOL
DRAPIER	Sylvain	PR1	Mécanique et ingénierie	SMS
FEILLET	Dominique	PR1	Génie Industriel	CMP
FEVOTTE	Gilles	PR1	Génie des Procédés	SPIN
FRACZKIEWICZ	Anna	DR	Sciences et génie des matériaux	SMS
GARCIA	Daniel	MR(DR2)	Génie des Procédés	SPIN
GERINGER	Jean	MA(MDC)	Sciences et génie des matériaux	CIS
GOEURIOT	Dominique	DR	Sciences et génie des matériaux	SMS
GRAILLOT	Didier	DR	Sciences et génie de l'environnement	SPIN
GROSSEAU	Philippe	DR	Génie des Procédés	SPIN
GRUY	Frédéric	PR1	Génie des Procédés	SPIN
GUY	Bernard	DR	Sciences de la Terre	SPIN
HAN	Woo-Suck	MR	Mécanique et ingénierie	SMS
HERRI	Jean Michel	PR1	Génie des Procédés	SPIN
KERMOUCHE	Guillaume	PR2	Mécanique et Ingénierie	SMS
KLOCKER	Helmut	DR	Sciences et génie des matériaux	SMS
LAFOREST	Valérie	MR(DR2)	Sciences et génie de l'environnement	FAYOL
LERICHE	Rodolphe	CR	Mécanique et ingénierie	FAYOL
LI	Jean-Michel		Microélectronique	CMP
MALLIARAS	Georges	PR1	Microélectronique	CMP
MAURINE	Philippe			CMP
MOLIMARD	Jérôme	PR2	Mécanique et ingénierie	CIS
MONTHEILLET	Frank	DR	Sciences et génie des matériaux	SMS
MOUTTE	Jacques	CR	Génie des Procédés	SPIN
NEUBERT	Gilles			FAYOL
NIKOLOVSKI	Jean-Pierre	Ingénieur de recherche		CMP
NORTIER	Patrice	PR1		SPIN
PIJOLAT	Christophe	PR0	Génie des Procédés	SPIN
PIJOLAT	Michèle	PR1	Génie des Procédés	SPIN
PINOLI	Jean Charles	PR0	Image Vision Signal	CIS
POURCHEZ	Jérémy	MR	Génie des Procédés	CIS
ROBISSON	Bruno	Ingénieur de recherche		CMP
ROUSSY	Agnès	MA(MDC)	Génie industriel	CMP
ROUSTANT	Olivier	MA(MDC)	Mathématiques appliquées	FAYOL
ROUX	Christian	PR	Image Vision Signal	CIS
STOLARZ	Jacques	CR	Sciences et génie des matériaux	SMS
TRIA	Assia	Ingénieur de recherche	Microélectronique	CMP
VALDIVIESO	François	PR2	Sciences et génie des matériaux	SMS
VIRICELLE	Jean Paul	DR	Génie des Procédés	SPIN
WOLSKI	Krzystof	DR	Sciences et génie des matériaux	SMS
XIE	Xiaolan	PR1	Génie industriel	CIS
YUGMA	Gallian	CR	Génie industriel	CMP

ENISE : Enseignants-chercheurs et chercheurs autorisés à diriger des thèses de doctorat (titulaires d'un doctorat d'État ou d'une HDR)

BERGHEAU	Jean-Michel	PU	Mécanique et Ingénierie	ENISE
BERTRAND	Philippe	MCF	Génie des procédés	ENISE
DUBUJET	Philippe	PU	Mécanique et Ingénierie	ENISE
FEULVARCH	Eric	MCF	Mécanique et Ingénierie	ENISE
FORTUNIER	Roland	PR	Sciences et Génie des matériaux	ENISE
GUSSAROV	Andrey	Enseignant contractuel	Génie des procédés	ENISE
HAMDI	Hédi	MCF	Mécanique et Ingénierie	ENISE
LYONNET	Patrick	PU	Mécanique et Ingénierie	ENISE
RECH	Joël	PU	Mécanique et Ingénierie	ENISE
SMUROV	Igor	PU	Mécanique et Ingénierie	ENISE
TOSCANO	Rosario	PU	Mécanique et Ingénierie	ENISE
ZAHOUANI	Hassan	PU	Mécanique et Ingénierie	ENISE

Mise à jour : 28/10/2014

UNIUNEA EUROPEANĂ
Fondul Social European POSDRU 2007-2013
Instrumente Structurale 2007-2013
MINISTERUL EDUCAȚIEI ȘI CERCETĂRII ȘTIINȚIFICE
OIPOSDRU
Universitatea POLITEHNICA din Bucuresti

Proiect cofinanţat din Fondul Social European prin Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007-2013
Investeşte în oameni!
Proiect EXCELDOC - POSDRU/159/1.5/S/132397
Excelență în cercetare prin burse doctorale și postdoctorale

UNIVERSITATEA **POLITEHNICA** DIN BUCUREŞTI
Facultatea de Automatică şi Calculatoare
Departamenul de Calculatoare

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE
Ecole Doctorale ED-SIS de Saint-Etienne
Laboratoire Hubert Curien, Institut Henri Fayol, Mines Saint-Etienne

Nr. Decizie Senat 239 din 03.12.2015

TEZĂ DE DOCTORAT

Un Web Social de Obiecte:

Interacțiune Autonomă și Flexibilă în Internetul Obiectelor

Weaving a Social Web of Things:

Enabling Autonomous and Flexible Interaction in the Internet of Things

Autor: Ing. Andrei-Nicolae CIORTEA

COMISIA DE DOCTORAT

Preşedinte	Prof. dr. Emil SLUȘANSCHI	de la	Universitatea „Politehnica“ din București
Conducător de doctorat-1	Prof. dr. Adina Magda FLOREA	de la	Universitatea „Politehnica“ din București
Conducător de doctorat-2	Prof. dr. Olivier BOISSIER	de la	École Nationale Superieure des Mines de Saint-Etienne
Co-supervizor de doctorat	Assoc. Prof. dr. Antoine ZIMMERMANN	de la	École Nationale Superieure des Mines de Saint-Etienne
Referent	Prof. dr. Costin BĂDICĂ	de la	Universitatea din Craiova
Referent	Dr. Fabien GANDON	de la	Institut National de Recherche en Informatique et en Automatique (INRIA)
Referent	Prof. dr. Erik MANNENS	de la	Universitatea din Gent
Referent	Assoc. Prof. dr. Alessandro RICCI	de la	Universitatea din Bologna
Referent	Prof. dr. Ștefan TRĂUȘAN	de la	Universitatea „Politehnica“ din București

Bucureşti 2015

To my grandfather

Acknowledgments

The past few years have been an incredible journey. This page is about the people that have supported me throughout this journey.

First of all, I would like to express my sincere appreciation and gratitude to my supervisors, Prof. Adina-Magda Florea, Prof. Olivier Boissier and Assoc. Prof. Antoine Zimmermann, for their constant support, guidance and feedback.

I would like to thank Prof. Adina-Magda Florea for introducing me to the wonderful field of artificial intelligence, for inviting me to join the AI-MAS laboratory, and especially for her constant trust and almost unlimited support in all my endeavors. Having had the opportunity to work with such an optimistic and supportive supervisor has completely altered the course of my life in the best possible way. In addition to this dissertation, an important result of our intense work over the past few years has been the Romanian Association for Artificial Intelligence, which is by any measure a significant achievement.

I would like to express my deepest gratitude to Prof. Olivier Boissier and Assoc. Prof. Antoine Zimmermann for their constant guidance and support in my research. Their insightful input and our interactions over the years have not only made me a better researcher, but also a better person. I also want to thank Prof. Olivier Boissier for making my one year and a half stay in Saint-Étienne possible and for undertaking many of the administrative burdens such that I could focus on my research instead. All these efforts have been much appreciated.

I also wish to thank my colleagues in the AI-MAS and ISCOD laboratories for their friendship and support. In particular, my friend Alex Sorici has been a constant companion throughout the years, in many travels and in many projects. I consider myself fortunate to have such a brilliant friend. Many thanks go out to Mihai Trăscău, Valentin Lungu, Costin Caval, Marius-Tudor Benea, Andrei Ismail and all my other colleagues in the AI-MAS team that I was fortunate to work with in our many projects. I want to thank Bissan Audeh, for being a most optimistic and positive office colleague, Reda Yaich, for our many discussions and his useful advices, and Amro Najjar, Oudom Kem, Marie-Line Barneoud, Gauthier Picard, Xavier Serpaggi, Niloufare Sadr, Nicolas Cointe and everyone in the ISCOD team for making my stay in Saint-Étienne most pleasant.

I would like to thank Cristian Stoica and AQUASoft for their financial support in the beginning of my PhD studies and for the wonderful experience of working together.

Finally, my deepest gratitude to my family and friends for their understanding and constant support in all my endeavors. I consider myself most fortunate to have so many meaningful relationships in my life.

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/132397. Andrei Ciortea has also been funded by the Rhône-Alpes region (France).

Abstract

The Internet of Things (IoT) aims to create a global ubiquitous ecosystem composed of large numbers of heterogeneous devices. To achieve this vision, the World Wide Web is emerging as a suitable candidate to interconnect IoT devices and services at the application layer into a Web of Things (WoT).

However, the WoT is evolving towards large silos of things, and thus the vision of a global ubiquitous ecosystem is not fully achieved. Furthermore, even if the WoT facilitates mashing up heterogeneous IoT devices and services, existing approaches result in static IoT mashups that cannot adapt to dynamic environments and evolving user requirements. The latter emphasizes another well-recognized challenge in the IoT, that is enabling people to interact with a vast, evolving, and heterogeneous IoT.

To address the above limitations, we propose an architecture for an open and self-governed IoT ecosystem composed of people and things situated and interacting in a global environment sustained by heterogeneous platforms. Our approach is to endow things with autonomy and apply the social network metaphor to create flexible networks of people and autonomous things. We base our approach on results from multi-agent and WoT research, and we call the envisioned IoT ecosystem the Social Web of Things.

Our proposal emphasizes heterogeneity, discoverability and flexible interaction in the IoT. In the same time, it provides a low entry-barrier for developers and users via multiple layers of abstraction that enable them to effectively cope with the complexity of the overall ecosystem. We implement several application scenarios to demonstrate these features.

Contents

Chapter 1

Introduction

In 1991, two independent events took place that would inspire future generations of researchers and developers. On August 6th, Tim Berners-Lee announced a summary of the World Wide Web project on the *alt.hypertext* newsgroup.[1] Soon to follow, the September issue of Scientific American published Mark Weiser's seminal article on *ubiquitous computing*, a vision in which networks of small computers disappear into the background and "weave themselves" into the physical world [Weiser 1991]. A couple of decades later, these two lines of research come to intersect.

Standardization efforts led by the Internet Engineering Task Force are rapidly turning Weiser's vision into reality [Ishaq 2013]. Low-cost devices with severely constrained resources can now converge at the network layer into an Internet of Things (IoT) [Montenegro 2007, Hui 2011, Shelby 2012]. At the application layer, the Constrained Application Protocol (CoAP) [Shelby 2014a] enables the direct integration into the Web of devices with as little as 100KiB of ROM and 10 KiB of RAM [Kovatsch 2015]. On account of its scalable architecture, the World Wide Web is emerging as the application architecture for the IoT, i.e. the so-called Web of Things (WoT) [Wilde 2007, Guinard 2010b]. In 2014, the World Wide Web Consortium chartered an interest group to investigate requirements for the standardization of a WoT.[2]

Integrating IoT devices into the Web enables the creation of an Internet-scale ecosystem of loosely coupled heterogeneous *devices* and *services*, henceforth *things*. This loose coupling between things facilitates mashing up services that extend to the physical world, henceforth IoT mashups or *physical mashups* [Guinard 2009], very much like mashing up regular Web services. Developers can then use standard Web technologies to mash up a Web-enabled wristband[3] with a Web-enabled coffee machine, for instance, such that there is always fresh coffee available when their owner wakes up in the morning, which significantly lowers the entry-barrier for the development of IoT applications.

1.1 Motivation

The Web enables application-layer interoperability in the IoT. Many cloud-based IoT platforms are already using the Web to provide data repositories for IoT de-

[1]http://www.w3.org/People/Berners-Lee/1991/08/art-6487.txt, Accessed: 30.11.2015.

[2]http://www.w3.org/2014/12/wot-ig-charter.html, Accessed: 30.11.2015.

[3]Such as a smart wristband from the Jawbone UP series: https://jawbone.com/up/, Accessed: 30.11.2015.

vices. However, in absence of standards to ensure interoperability at the platform level, the WoT is already evolving towards large *silos of things* [Blackstock 2014a]. Existing WoT platforms are *closed* and expose *heterogeneous* application programming interfaces (APIs). By analogy, a similar situation is presented by the problem of walled gardens in the Social Web [Halpin 2010].

Limitation 1. Things are confined to Web silos.

Furthermore, many existing WoT platforms do not facilitate the discovery of things. Any relations among things, such as the one between the wristband and the coffee machine in our previous example, are hard-coded into the application logic. The lack of an external and uniformly accessible representation of relations among things hinders the "network effect".

The WoT initiative encourages interconnecting things via hyperlinks such that they become discoverable [Guinard 2010b]. However, with predictions of billions of connected devices by the end of 2020 [MacGillivray 2013], things have to be interconnected in an effective and consistent manner.

Limitation 2. Things are not discoverable in an effective and consistent manner.

The WoT facilitates the development of IoT mashups via standard Web technologies. Furthermore, mashup editors[4] [Blackstock 2012, Kleinfeld 2014] can further lower development costs or even enable tech savvy users to create their own IoT mashups. However, manually "wiring" the IoT does not scale. Furthermore, once created, these mashups are static and cannot adapt to dynamic environments or evolving user requirements [Mayer 2014c].

Limitation 3. Manually created IoT mashups are static and do not scale.

Another well-recognized challenge in the IoT is enabling people to manage and interact with large numbers of heterogeneous things [Randall 2003, Formo 2012, Takayama 2012], and to coordinate and keep track of interactions between collaborative things [Brush 2011, Formo 2012, Mayer 2014a].

Limitation 4. Managing, interacting with and keeping track of large numbers of heterogeneous things is cumbersome.

Our thesis is that we can address the above limitations by endowing things with *autonomy* and applying the *social network metaphor* to the IoT to create flexible networks of people and *autonomous things.*

1.2 Research Questions

The overarching objective of this dissertation is to contribute to the development of a global IoT ecosystem in which people and *autonomous things* interact in a *flexible*

[4]http://www.nodered.org/, Accessed: 30.11.2015.

fashion. We build our approach on state-of-the-art results from multi-agent and WoT research and we call the envisioned IoT ecosystem the *Social Web of Things (SWoT).* To bring about this ecosystem, we address four research questions, which we present in what follows. The first two questions are focused around the core tenets of our thesis and the last two are focused around addressing the identified limitations.

Research Question 1. How can we bring systems of autonomous things to the Web of Things?

The first cornerstone of our thesis is autonomy. Endowing *things* with autonomy raises new challenges to be addressed, such as how can control be enabled over the autonomy of things, or how can developers design and program autonomous things and networks of autonomous things. To this purpose, we adapt models from multi-agent research and propose an architecture for the SWoT that introduces several layers of abstraction in order to enable developers and users to cope with the overall complexity of the envisioned ecosystem.

Research Question 2. How can we model networks of people and autonomous things such that things can manipulate and reason upon them?

The second cornerstone of our thesis is to apply the social network metaphor to the IoT to create flexible networks of people and autonomous things, which we call *socio-technical networks (STNs).* STNs are the building blocks of the SWoT and span across the physical-digital space: they reflect the physical world via sensors and can reflect back on the physical world via actuators. STNs should be *flexible* in the sense that both people and things should be able to manipulate them in a *reliable fashion.* Reliability implies that both people and things should be able to interpret and reason upon STNs. To this purpose, we introduce and formalize concepts that take into account the various dimensions and requirements for STNs.

Research Question 3. How can we enable things to transcend Web silos?

If the SWoT is to be a global ecosystem, things must not be confined to Web silos (cf. Limitation 1). Furthermore, in addition to WoT platforms, it is worth to note that the SWoT ecosystem could also benefit from many other existing Web platforms, such as social platforms. To this purpose, we provide solutions to hide platform heterogeneity behind uniform interfaces such that things can access and use *heterogeneous platforms* in a *uniform fashion.*

Research Question 4. How can we enhance *discoverability* and *flexible interaction* in the Web of Things?

Things in the SWoT should be discoverable (cf. Limitation 2). Furthermore, they should be able to autonomously discover and interact with one another. In other words, IoT mashups should be able to "rewire" themselves in order to adapt to dynamic environments and evolving user requirements (cf. Limitation 3). In a similar manner, people should be able to discover and interact with *heterogeneous things* in a *uniform fashion* (cf. Limitation 4). We consider the last two aspects to be similar in the sense that if large numbers of people and heterogeneous things are to interact with one another in a scalable and flexible fashion, they must be "decoupled" by means of a *uniform interaction mechanism.* To this purpose, we propose to use our STN model and existing programming paradigms for multi-agent systems.

1.3 Dissertation Outline

This dissertation is structured in four parts:

In Part I, we analyze the state-of-the-art in search for related models and technologies that can be used to bring about the envisioned IoT ecosystem. In Chapter 2, we discuss the architecture and various facets of the World Wide Web. We define in further detail the problem of transcending Web silos and discuss current developments in the field on which we base our approach.

In Chapter 3, we look at emerging paradigms in the WoT in order to define and discuss in further detail the limitations that motivate our work. In this chapter, we also discuss related work that investigates the use of social aspects in the IoT/WoT, in particular social networks and similar concepts.

In Chapter 4, we discuss models and results from multi-agent research that can be used to enable autonomy, sociability and regulation in the SWoT.

In Part II, we introduce our proposed architecture, models and solutions for the development of the envisioned IoT ecosystem. In Chapter 5, we introduce several application scenarios to help illustrate our vision and further define the properties and requirements for the SWoT. Following these requirements, we present the foundational principles on which we build our approach and propose a layered architecture for the SWoT. In this chapter, we address Research Question 1.

In Chapter 6, we define in further detail the various dimensions of an STN and formalize our discussion to provide a general mathematical model for STNs. Things can use this model to obtain an unambiguous representation of an STN and of the operations through which they can participate in the STN. In this chapter, we address Research Question 2.

In Chapter 7, we apply our STN model to propose solutions for the integration of heterogenous platforms into the SWoT. We propose a progressive integration strategy that balances platform design and implementation autonomy versus alignment with the ecosystem. In this chapter, we address Research Question 3.

In Part III, we present the current validations of our work. In Chapter 8, we demonstrate that our approach supports platform heterogeneity by deploying a SWoT environment that integrates several existing social platforms, namely Facebook, SoundCloud, and Twitter, and a WoT platform, that is Dweet.io. We also present and integrate into this SWoT environment our own implementation of an

STN platform that conforms to all the requirements of our progressive integration strategy in Chapter 7.

In Chapter 9, our investigation comes full circle: we present and discuss implementations of the application scenarios introduced in Chapter 5. In doing so, we validate the foundational principles that provide the underpinning of our proposal and we demonstrate that we are able to successfully apply existing multi-agent technology to facilitate the development of SWoT applications. In this chapter, we address Research Question 4.

In Part IV, we provide a summary of our work and highlight directions for future research.

Part I

State of the Art

Chapter 2

A Hitchhiker's Guide to the World Wide Web

Contents

The World Wide Web exhibits several architectural properties that have made it a suitable candidate to interconnect Internet of Things (IoT) devices and services at the application layer, such as *scalability* and *evolvability*. It is thus desirable to preserve these architectural properties in our envisioned IoT ecosystem. To this purpose, in this chapter we discuss the principles that provide the underpinning of the Web architecture. We also "hitchhike" through the various facets of the Web in order to identify and discuss current developments that could be useful to bring about the envisioned IoT ecosystem.

In Section 2.1, we discuss the architecture of the Web and investigate in further detail the problem of Web silos (cf. Limitation 1). In Section 2.2, we discuss the state of the Social Web and future directions envisioned by the World Wide Web Consortium. In Section 2.3, we discuss the Web as an information space for machines. In Section 2.4, we discuss recent developments that enable the integration of physical devices into the Web.

2.1 The Architecture of the Web

The development of the modern Web was guided by the *Representational State Transfer (REST)* architectural style [Fielding 2000]. We briefly present REST in Section 2.1.1. In Section 2.1.2, we use REST as a theoretical framework to analyze the problem of Web silos (see Limitation 1).

It is worth to note that there exists an official, standard specification of the architecture of the Web [Jacobs 2004]. However, we choose to focus our discussion on REST because (i) it defines the principles underlying the Web architecture, and (ii) it allows us to keep the discussion at a higher level of abstraction and independent of the Web.

2.1.1 Representational State Transfer

REST is an architectural style for distributed hypermedia systems that emphasizes "scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems" [Fielding 2000].

2.1.1.1 REST in a nutshell

In a system that conforms to the REST architectural style, henceforth called *RESTful*, components follow a *request/response* interaction model. Interaction between components is *data-driven*: they exchange *representations* of *resources* by means of a *small set of generic methods* with *well-defined semantics*. All resources have *uniform identifiers* and can thus be referenced globally, independent of context. Interaction between components is *stateless*, which improves *scalability* and enables the use of *intermediary components*.

A feature that is central to REST is having a *uniform interface* between components. The purpose of having a uniform interface is to hide component-specific implementation details. Components are thus *loosely coupled*, which allows them to be deployed and to evolve independently from one another. This uniform interface is achieved by means of *uniform identifiers*, *standard methods*, *standard representation formats*, and *hypermedia-driven interaction*.

2.1.1.2 The uniform interface constraint

REST is defined as a *coordinated* set of architectural constraints. Applied as a whole, these constraints induce the properties discussed above. In what follows, however, we focus our discussion on a single constraint, that is the *uniform interface constraint*, which is central to the REST architectural style, our approach and the rest of this dissertation. The other REST constraints are described in detail in [Fielding 2000].

The *uniform interface constraint* is defined by four interface constraints, that is [Fielding 2000]: *identification of resources*, *manipulation of resources via repre-*

sentations, *self-descriptive messages*, and *hypermedia as the engine of application state*.

Identification of resources. *Resources* are key abstractions in REST, and anything that is worth being referred to is a resource. Resources have *uniform identifiers*, and resource identifiers cannot be reassigned. For instance, Web resources are identified by means of Uniform Resource Identifiers (URIs) [Berners-Lee 2005], and the naming authority that assigns a URI is responsible for maintaining its semantic validity over time.

It is worth to emphasize the importance of having *globally identifiable resources*.

Manipulation of resources via representations. A resource is an abstract entity that can have multiple representations in different formats (e.g., HTML, JSON, XML). This interface constraint states that a client does not have direct access to resources, it can only send and receive representations of resources to and from a server. The representation of a resource may represent the current state of a resource or the desired state of the resource. The difference is made by the control data within a message, which leads to the next interface constraint.

Self-descriptive messages. A message must include all necessary metadata for describing its meaning, which includes *methods* and representation formats, a.k.a. *media types*.

Methods used to invoke the message must be standard and agreeable between the client, server and any intermediaries in between. For illustrative purposes, well-known HTTP methods include [Fielding 2014c]:

- `GET`: retrieve a representation of the state of a resource;
- `POST`: create a new resource;
- `PUT`: replace the state of a resource using the representation enclosed in the request payload; `PUT` may also be used to create a resource with a preferred URI;
- `DELETE`: remove a resource;
- `OPTIONS`: return the methods allowed on a resource.

`GET` and `OPTIONS` are *safe* methods, i.e. they have no side effects. `GET`, `PUT`, `DELETE` and `OPTIONS` are *idempotent* methods, meaning that multiple requests have the same effect as a single request. Responses to `GET` requests are also *cacheable*.[1]

Hypermedia as the Engine of Application State (HATEOAS). The last interface constraint states that interaction has to be *driven by hypermedia*. The HATEOAS constraint is central to achieve a uniform interface.

[1]The response to a `POST` request may also be cacheable if the request contains explicit freshness information, however `POST` caching is not widely implemented [Fielding 2014c]. This was one of the initial drawbacks of WS-* Web services, which rely heavily on `POST` requests.

To best illustrate hypermedia-driven interaction, we use as an example an HTML-based application composed of multiple interlinked Web pages. This application is, in fact, a finite state machine, in which each page represents a state and hyperlinks between pages represent transitions between states. Given an entry URI into this application (i.e., the URI of one of the pages), a client (e.g., a browser) can dereference the URI to retrieve an *HTML representation* of the *page* it identifies. This action triggers a transition to a new state, and if this transition is completed successfully the client can now choose from a new set of reachable states. The *controls* required to transition to new states are encoded in the HTML representations retrieved from the origin server.

Therefore, in each state of an application, a client can choose a next reachable state from a selection of states retrieved from an origin server. The information needed to transition to new states has to be conveyed to the client via hypermedia. Given an entry URI, the client should not require any additional information besides standard interaction protocols and a set of standard media types.

Hypermedia-driven interaction is what ensures connectivity on the Web and enables crawling across websites. However, most existing Web application programming interfaces (APIs) are non-hypermedia interfaces, which leads us to the problem of Web silos (see Limitation 1).

2.1.2 The "out-of-band information" problem

From a technical perspective, we can now identify the root cause of having isolated Web platforms (see Limitation 1): most existing Web platforms expose non-hypermedia APIs that *violate the uniform interface constraint* [Fielding 2008, Webber 2010]. Common characteristics of these APIs include:

- the use of platform-specific identifiers for resources;
- the use of HTTP [Fielding 2014c] in a non-standard manner, such as creating resources via `HTTP GET` or deleting resources via `HTTP POST`;
- the use of representations that expose platform-specific data models via generic media types; for instance, most existing APIs produce JSON-based [Bray 2014] representations that are processed based on information provided via the APIs' documentation;
- they do not use hypermedia to drive interaction with clients, which implies that clients typically have to hard-code URIs or URI templates.

Consequently, in order to interface with these APIs, clients must hard-code platform-specific knowledge that is typically available via *out-of-band documentation*, and are thus *tightly coupled* to the platforms. Throughout the rest of this dissertation, we generally refer to this *platform heterogeneity* problem as "the out-of-band information problem".

There are, in fact, three approaches commonly used for the development of Web APIs [Webber 2010]:

- *RPC over HTTP*: these APIs typically expose a single endpoint, and clients interact with the API via `HTTP POST` requests that enclose in the request body all the information required to interpret an invoked *operation*; HTTP is therefore used only as a transport protocol;

- *URI tunneling*: clients typically interact with these APIs by encoding and transmitting in the request URI an *operation* to be performed (e.g., via path parameters) and any parameters required to perform the operation (e.g., via query parameters); these APIs typically rely only on `HTTP GET` and `HTTP POST` requests;

- *CRUD over HTTP*: clients typically interact with these APIs by exchanging representations of resources using a *Create, Read, Update, Delete (CRUD)* interaction pattern via `HTTP POST`, `HTTP GET`, `HTTP PUT`, and `HTTP DELETE`, respectively; HTTP is thus used here as an application protocol.

The first two approaches therefore use a *control-driven interaction pattern* (i.e., the APIs are defined in terms of operations invoked remotely by clients). The latter uses an approach that is somewhat closer to data-driven interaction, however clients still have to hard-code *operations* defined via the APIs' documentation as an *(HTTP method, URI template)* tuple and a set of required parameters, which is essentially more similar to a control-driven interaction pattern. It is worth to note that most existing APIs generally use one of the last two approaches, and in many cases a mix of the two.

Nevertheless, having a non-uniform, non-hypermedia API that is driven by out-of-band information is not a problem in and of itself. The advantage of non-hypermedia APIs is that they are simple and intuitive for most developers, which motivates their success on the Web. Public Web APIs also tend to have a low change frequency and generally use API versioning to ensure that existing clients using the API do not break if the API changes. In the context of our work, however, non-uniform APIs hinder the development of the envisioned IoT ecosystem. In order to free things from Web silos, it is thus necessary to provide solutions to achieve uniform interfaces for heterogenous APIs.

The out-of-band information problem is relevant to our research endeavor for two reasons. First, it provides a better understanding of the integration challenges that have to be addressed in order to integrate existing Web platforms into the envisioned IoT ecosystem. Second, it helps identify a situation that should be avoided for the development of new platforms that are to support this ecosystem.

2.2 The Web of People

The problem of Web silos is commonly known in the Social Web as *the problem of walled gardens* [Halpin 2010]. In this section, we look for existing approaches and technologies that could be useful to address this problem.

We discuss the problem of walled gardens in more detail in Section 2.2.1. In Section 2.2.2, we briefly summarize the vision for an open and distributed Social Web as presented in the final report of the W3C Social Web Incubator Group [Halpin 2010]. We discuss enabling technologies that could help achieve that vision in Section 2.2.3.

2.2.1 The problem of walled gardens

The present-day Social Web is governed by a small number of social platforms with high user adoption, such as Facebook or Twitter. A social platform often hosts a user's personal data and social graph, and provides a set of features to facilitate and manage interactions with other users. It is common, however, that once a user has entered his personal data on a social platform, he may access it only through proprietary interfaces. His data is not portable, he cannot move it to a different platform, and he cannot directly interact with users on other platforms. Users who intend to use multiple social platforms thus have to create and manage social graphs on each of these platforms. Advocates of an open Social Web use e-mail as an analogy to highlight the drawbacks of this state of affairs: if one would be able to use a given e-mail address to correspond only with users from the same e-mail provider, the utility of e-mail services as a whole would be significantly less.

From a developer's perspective, creating applications that run on multiple social platforms is not straightforward. Like most Web APIs, the APIs of social platforms are heterogeneous (see Section 2.1.2), and the functionality they support varies greatly. For instance, while the Twitter API supports creating and deleting connections among users, the Facebook API does not. In lack of standard media types for their particular application domain, social platforms produce representations that expose platform-specific data models. Social platforms remain huge isolated data silos. The final report of the W3C Social Web Incubator Group notes that "a truly universal, open, and distributed Social Web architecture is needed" [Halpin 2010].

2.2.2 An open and distributed Social Web

In what follows, we summarize the vision of the W3C Social Web Incubator Group for a standards-based, open and distributed Social Web [Halpin 2010]. Openness is one of the general requirements for our socio-technical overlay, and thus this vision, which is the result of a community effort, is highly relevant to our work. The focus of our interpretation is on the core concepts, how they relate to one another and how the open Social Web should work. We discuss enabling technologies in Section 2.2.3.

In this vision, a *user* may be "a person, organization, or other agent that participates in online social interactions on the Web" [Halpin 2010]. It is worth noting that this definition is generic enough to include things as well. A user is described through *attributes* (e.g., name, e-mail), which are grouped in *profiles*. A user may hold several such profiles, and the same attribute may be shared among multiple profiles. Once an attribute is modified, it is synchronized across all profile instances it is a part of. Some attributes may be dynamic by nature (e.g., geolocation).

Attributes and social connections within a profile may be distributed across multiple *social platforms*. A social platform provides users with features that enable them to build and manage their social graphs, publish or consume social media, or use *social apps*. A social platform may be hosted by a third party, but it may also be owned and controlled by a user. The democratization of the Social Web is central to this vision: users may choose to run their own nodes and install social apps locally, without having to make a compromise between the social platform they use and the social apps that are available to them. In a standards-based Social Web, social apps would be, at least to some extent, agnostic to the underlying social platform. Therefore, an open Social Web would foster the development of social apps and enable developers to focus on providing added value services to users rather than on features for managing social structures.

The definition of a *social connection* in this vision is broad: they are "associations between a profile and a resource (or group of resources)" [Halpin 2010]. Therefore, social connections may be established between users (e.g., friendship), but also between users and social media items (e.g., likes). A social connection may be uni-directional or bidirectional, and there may be multiple connections between the same two users through several of their profiles. It is social platforms that support specific types of connections, while social apps "make, maintain and expand these connections" [Halpin 2010].

One of the central characteristics of social connections in an open Social Web is that they are portable. They are not confined to a particular platform and users are not required to re-establish connections on a different platform.

Another concept we are interested in is the one of *social group*. Social groups are defined as "named groups of resources" [Halpin 2010], such as a group of friends or a favorite list of movies. Therefore, no strong assumptions are made about groups. They could be membership-based groups, such as Facebook groups, but also collections of users, such as Twitter lists.

In summary, in our interpretation, the open Social Web vision adds a social overlay to the Web. This overlay is collaboratively built and maintained by social platforms. Therefore, regardless of being decentralized (e.g., Diaspora) or centralized (e.g., Twitter, Facebook), social platforms need to be integrated in the same social overlay to achieve this vision. Users may choose to use a social platform they trust, but they may also run their own platform without being cut off from social apps or users on other platforms. Users may also easily move their personal and social data from one platform to another, without having to go through the cumbersome process of re-entering it. Social apps are built on top of this social overlay. A social app is not confined to one platform and it may even extend its functionality across multiple platforms. In Part II, we build upon and extend this interpretation to include things as first-class citizens of a socio-technical overlay for the WoT.

2.2.3 Enabling technologies

Tim Berners-Lee argued that the technologies for building a distributed Social Web are already available and proposed "socially aware cloud storage", which separates social applications from storage [Berners-Lee 2009]. This position is further developed in [Yeung 2009]. The W3C Social Web Incubator Group's final report offers a thorough overview of the Social Web in 2010 and technologies that might enable a standards-based, open and distributed Social Web [Halpin 2010].[2] In this section, our intention is not to provide an exhaustive list of existing technologies that are relevant to implementing this vision. The purpose of our discussion is to introduce technologies, and their limitations, that are relevant to the rest of this thesis.

We search for solutions to three fundamental technical challenges raised by the open Social Web vision. We begin with a critical security-related question: how can users access resources, or grant access to resources, in an open and decentralized Web? Web-scale mechanisms for enforcing security are essential. Second, how can resources be effectively distributed across multiple social platforms? Answers to these two questions should provide solutions to build a secure and open Web of resources. The last question we are interested in is: what are existing approaches to support interoperability among social platforms, thus avoiding the problem of walled gardens?

2.2.3.1 Identification, authentication and authorization

In the open Social Web vision, users may access resources, or grant others access to resources, regardless of the underlying social platform. A critical problem that needs to be addressed is thus being able to identify and authenticate users, and to authorize third-party access to resources.

HTTP provides a challenge-response authentication framework [Fielding 2014a]: a server may challenge a client when a restricted resource is requested, and the client may respond with authentication information. This simple framework, however, does not provide support for allowing third-party access to resources. The client would have to share his credentials with the third party, which is obviously inconvenient.

OAuth [Hardt 2012] addresses this issue by adding an authorization layer which separates the client from the resource owner. The client may send an authorization request to the owner of a restricted resource. If authorization is granted by the owner, the client may then request an access token from an authorization server. The access token typically includes specific attributes, such as scope or lifetime. Using this access token, the client may then request the protected resource from a resource server.[3] When the access token becomes invalid or expires, it may be refreshed.

[2]The report also covers several distributed social networking platforms, such as Diaspora (http://diasporafoundation.org) or Appleseed (http://github.com/appleseedproj).

[3]The authorization server and the resource server may be one and the same.

OAuth is the de facto industry authorization framework. Most social platforms, including Facebook and Twitter, use OAuth. However, there is a limitation which is significant in the context of an open Social Web: OAuth was not designed with a focus on interoperability. OAuth is a rich and highly extensible authorization framework. The specification includes many optional components, while "a few required components are partially or fully undefined". Section 1.8 of RFC 6749 notes that "without these components, clients must be manually and specifically configured against a specific authorization server and resource server in order to interoperate". For instance, a social thing would have to be manually configured against a social platform in order to access a protected resource on that platform.

In February 2014, the OpenID Foundation launched OpenID Connect 1.0, which adds an identity layer on top of OAuth 2.0 [Sakimura 2014]. Through OpenID Connect, a client may verify the identity of a user via an authorization server that authenticates the user. The server, also called an OpenID Provider (OP), securely returns the result of the authentication to the client, also called a Relying Party (RP). The RP may also obtain basic profile information about the user from the OP. The major benefit brought by the OpenID Connect standard is that any compliant RP may authenticate users through any compliant OP. Therefore, the specifications create a decentralized single sign-on system, one in which users may choose from a variety of OPs. It is worth mentioning that some social platforms, such as Facebook or Twitter, are also identity providers running on OAuth, however they provide proprietary services and do not interoperate with other platforms.

Another mechanism for identification and authentication is described by the WebID Authentication Protocol [Sambra 2015b]. WebID makes use of TLS and client-side certificates for verifying the identity of a client. When a client requests a protected resource, the server asks for an X.509 certificate [Cooper 2008]. In addition to a public key, this certificate also contains the user's WebID, which is a URI identifying the user. By dereferencing the URI, the server returns the WebID profile document at the given location. This document also includes a public key. If this key matches the public key in the certificate, it implies that the client who established the TLS connection has the matching private key. Therefore, it is assumed that the client was delegated by and is acting for the owner of that WebID. The WebID profile uses the Friend-of-a-Friend (FOAF) vocabulary to describe basic profile information about its owner (e.g., name, social connections).

The advantage of WebID is that it is fairly simple and efficient, for which reason it caught some traction at W3C. For instance, Tim Berners-Lee proposed WebID as the default single sign-on system for socially aware cloud storage [Berners-Lee 2009]. However, WebID is yet to become a W3C recommendation. The main drawback is its user experience and potential for misuse: while the protocol itself works elegantly and does not require passwords, it requires users to manage certificates on the client-side, which provides a poor user experience in most browsers and has even less support on mobile devices [Halpin 2010]. Without buy-in from browser manufacturers, further adoption of WebID seems to stall. In addition, WebID tends to tie users to browsers, and thus to the devices they use. Technical workarounds for these issue may be conceived with additional cloud services, but this might cost WebID any edge over alternatives such as OpenID Connect. Nonetheless, while WebID is still to provide a good user experience for people, this would not be a problem for social things. Therefore, we believe WebID is a good candidate for verifying the identity of things in the SWoT. Furthermore, it could also work in conjunction with OpenID Connect, which might be a better choice for human users.

2.2.3.2 Profiles and social media

Global identification mechanisms and single sign-on systems enable users to securely access and share resources on the Web. Another central aspect of the open Social Web vision is distributing profiles and social media across social platforms and apps. In this section, we cover some vocabularies that may serve this purpose. We discuss semantics on the Web in Section 2.3.

Friend-of-a-Friend (FOAF) [Brickley 2014] is a vocabulary for describing social networks. FOAF defines concepts and properties that may be used to describe people, groups, organizations and other entities, and how they relate to one another. For instance, a FOAF description may include basic profile information about a person (e.g., name, e-mail, gender) and uni-directional social connections to other known persons. These descriptions may be embedded in HTML pages or published as standalone documents. Anyone may publish FOAF descriptions. Furthermore, given that FOAF uses URIs to identify resources, any description may link to other resources on the Web, thus building a Web-scale distributed social graph.

The FOAF project was launched in 2000[4], before the rise of the Social Web. Its main use case at the time was publishing descriptions on personal Web pages. While FOAF does not provide functionality in itself, it provides a standard way of describing social networks. There are several social platforms that generate FOAF profiles and some extensions exist for the ones that do not [Halpin 2010]. It is important to note that FOAF provides a simple and generic model for social networks of people, and cannot be used to describe social connections with or among things. However, FOAF may be easily extended and used alongside other vocabularies.

One such vocabulary is Semantically-Interlinked Online Communities (SIOC) [Bojars 2010]. Many terms in SIOC are defined with reference to FOAF. While FOAF describes networks of people, SIOC describes the content they generate on the Web. Content is described in terms of forums and threads, with posts that may reply to previous messages. Content is created through user accounts that may enact different roles, for instance in forums. One relation modeled through SIOC, which seems especially important in the SWoT, is the hosting of data. In addition, extensions of the core module allow attaching access rights to roles and tags/categories to posts.

Similar to FOAF, the SIOC project was launched in the early days of the Social Web (2004).[5] While its simple and generic model enables the structuring of user-

[4]http://www.foaf-project.org/, Accessed: 04.11.2015
[5]http://sioc-project.org/, Accessed: 04.11.2015

generated content, the vocabulary may also be extended further to describe social media content in a fashion closer to the one of modern social platforms. It is worth to note that SIOC is one of the core vocabularies used in Drupal 7.[6]

2.2.3.3 Interoperability

The problem of walled gardens reflects on both users and developers: users are confined to one social platform or the other, and developers are required to integrate their social apps with each individual platform. In the open Social Web vision, users on different social platforms may interact unrestricted, while developers build social apps that are decoupled from the underlying social overlay and whose functionality may extend across multiple social platforms. To achieve this vision, it is necessary for social platforms to interoperate. Profiles and social media vocabularies, such as FOAF and SIOC, provide model-level agreement. Going a step further, some API-level interoperability is also required.

OpenSocial[7] is an open standard that tackles the problem of decoupling applications from social platforms. It provides a collection of JavaScript APIs that offer standard access to social data. Any compliant social app, also referred to as a gadget, may run on any compliant social website, also called a container. The standard covers a broad set of functionalities, such as retrieving information about a person or list of persons, creating and deleting connections, publishing and consuming social media items etc. Gadgets, however, run within the context of a container. For instance, while running a gadget on one social platform, the user cannot interact with friends on a different platform.

OpenSocial was launched in 2007 by Google, together with MySpace and other social platforms.[8] Security issues with the early release were not very encouraging for developers.[9,10] OpenSocial has yet to see wide adoption. As of January 1st, 2015, the OpenSocial standardization effort moved to the W3C Social Web Working Group.[11]

Following our discussion, we conclude that the technologies for building a standards-based, open and distributed Social Web are now available. However, the problem of walled gardens goes well beyond the technical challenges. The lack of standards that would guide the development of the Social Web has lead to the rise of a small number of closed platforms with huge user adoption. Furthermore, each of the established social platforms appears to hold monopoly in a well defined category: Facebook as a general-purpose social network, LinkedIn for professional use, Twitter for disseminating information etc. There seems to be little incentive for existing

[6]https://www.drupal.org/project/sioc, Accessed: 09.11.2015.

[7]http://opensocial.github.io/spec/2.5.1/Core-API-Server.xml, Accessed: 09.11.2015.

[8]http://googlepress.blogspot.fr/2007/11/google-launches-opensocial-to-spread_01.html, Accessed: 09.11.2015.

[9]http://techcrunch.com/2007/11/02/first-opensocial-application-hacked-within-45-minutes/, Accessed: 09.11.2015.

[10]http://techcrunch.com/2007/11/05/opensocial-hacked-again/, Accessed: 09.11.2015.

[11]http://www.w3.org/blog/2014/12/opensocial-foundation-moves-standards-work-to-w3c-social-web-activity/, Accessed: 09.11.2015.

social platforms to tear down the walls around their users. But while there might be little change foreseen in the Social Web landscape, with the right tools already available and an estimated 25 billion connected things by 2020[12], openness is an essential requirement for achieving the full potential of a Social Web of Things.

2.3 The Web for Machines

In his seminal 2001 paper [Berners-Lee 2001], Tim Berners-Lee presents a vision for a future Web in which agents assist people with everyday tasks. Berners-Lee called this vision the Semantic Web. The challenge is, however, that the Web looks very different for a machine than it does for a human.

In order to consume Web content in a reliable fashion, machines need structured data (see Section 2.3.1). Going a step further, if they are also to autonomously reason upon Web content, they need semantic data (see Section 2.3.2). Finally, if they are also to manipulate Web content, machines need to be able to read and write semantic data to the Web in a reliable fashion (see Section 2.3.3). In this section, we look at existing technologies that could enable things to autonomously produce and consume information in a Web-based ecosystem.

2.3.1 Adding structure to information

The standard for adding metadata to the Web is the Resource Description Framework (RDF) [Cyganiak 2014]. RDF provides a simple data model to describe resources through statements, with several serialization formats being available. Originally designed to describe Web resources, RDF is domain-independent and may be used to represent any knowledge that can be encoded as a graph.

An RDF triple is a structure of the form `<subject> <predicate> <object>`, a.k.a. a *triple*. The subject is either an IRI or a blank node, the predicate is an IRI, and the object is an IRI, a blank node, or a literal, that is a basic value, such as a string, number or a date. Multiple RDF triples form an RDF graph. Data stored in RDF graphs can be retrieved and manipulated with RDF query languages, such as SPARQL.

RDF has several properties we are particularly interested in. It is a graph-based data model. Merging sets of triples is a cheap operation. Furthermore, IRIs enable the representation of distributed graphs in RDF, but also the interlinking of various datasets and thus cross-dataset queries.

2.3.2 Web ontologies

RDF adds metadata to resources. Metadata by itself, however, does not add semantic information. Going a step further, ontologies provide agreement on the semantics of IRIs and the structure of the metadata. It is this agreement that enables knowledge exchange across applications.

[12] http://www.gartner.com/newsroom/id/2905717, Accessed: 26.01.2015.

In a widely accepted definition, an ontology is “a specification of a conceptualization” [Bruber 1993]. In other words, an ontology provides a formal description of concepts and how they relate to one another. Breslin et. al. point out that most approaches to ontology modeling define [Breslin 2009]:

- a distinction between classes and instances, where a class is a set of instances; a binary relation denotes that an instance is an element of a class, and there is usually a subclass partial order over the set of classes;

- a set of properties, also called attributes, as binary relations usually having a certain domain and range.

RDFS [Brickley 2004] is a vocabulary that defines primitives to describe classes, instances and properties. There are several well known vocabularies originally defined in RDFS, such as FOAF (cf. Section 2.2.3.2). The expressive power of RDFS, however, is limited. For instance, RDFS cannot be used to say that a property is transitive. More expressive semantic modeling of RDF data is provided by OWL.

OWL [W3C OWL Working Group 2012] is an ontology language with formally defined semantics based on Description Logics. OWL provides axioms to define characteristics and constraints of classes and properties, such as any person has a mother and the maternal relationship may exist only between persons. OWL has also an RDF vocabulary and may be used in conjunction with RDFS. Several sublanguages of OWL are available, with different computational properties.

2.3.3 Reading/Writing Linked Data

In what follows, we discuss in detail the Linked Data Platform (LDP) to gain greater insight into the interaction patterns it promotes. The LDP defines a protocol for reading and writing *linked data* [Berners-Lee 2006] on the Web. It provides conventions for *information resources* [Jacobs 2004] and rules for HTTP operations [Fielding 2014b] on those resources.

2.3.3.1 LDP Resources and LDP Containers

LDP Resources [Speicher 2015] are information resources that conform to the lifecycle patterns and conventions defined by the LDP. An LDP server [Speicher 2015] may manage two types of LDP Resources: those whose state is fully represented as an RDF graph [Cyganiak 2014], referred to as LDP RDF Sources [Speicher 2015], and those whose state is represented in any other format, such as HTML documents and binary files, referred to as LDP Non-RDF Sources [Speicher 2015].

A specific type of LDP RDF Sources are LDP Containers [Speicher 2015], which represent collections of LDP Resources. LDP Containers are used to organize the overall space of information and manage the lifecycle of LDP Resources.

2.3.3.2 HTTP operations

When serving LDP Resources, the LDP requires conformant servers to support several features that are otherwise optional in HTTP.

The HTTP `POST`, `PUT` and `DELETE` methods [Fielding 2014c], and the HTTP `PATCH` method [Dusseault 2010], are optional. If LDP servers support these methods, the LDP imposes new requirements in addition to existing standards. We discuss these additions briefly in what follows.

Conforming LDP clients should create LDP Resources via `POST` requests to known LDP Containers. LDP servers may allow clients to suggest the URI of the resource to be created using the `Slug` header field [Gregorio 2007]. LDP servers may also accept resource creation via `PUT` requests. When a resource is created successfully, LDP servers are required to add the appropriate containment and membership triples (see [Speicher 2015] for more details), and are required to send an HTTP response with status code `201 Created` and the `Location` header field [Fielding 2014c] set to the URL of the created resource.

LDP severs may support the HTTP `PUT` method for updating LDP Resources. LDP servers are required to replace the entire persistent state of the resource with the entity enclosed in the request body.

LDP servers may support the HTTP `DELETE` method for deleting LDP Resources. When deleting a resource in an LDP Container, LDP servers are required to remove corresponding containment and membership triples.

LDP servers are required to support the HTTP `GET`, `HEAD` and `OPTIONS` methods [Fielding 2014c] for all LDP Resources. In addition, LDP servers are required to include the following header fields in response to HTTP `GET` and `OPTIONS` requests:

- the HTTP method tokens supported for the requested resource in the HTTP response header `Allow`;
- if the HTTP `POST` method [Fielding 2014c] is supported for the requested resource, include accepted representation formats in the HTTP response header `Accept-Post` [Speicher 2015];
- if the HTTP `PATCH` method [Dusseault 2010] is supported for the requested resource, include accepted representation formats in the HTTP response header `Accept-Patch` [Dusseault 2010].

It is worth to note that, per RFC 2731, conforming HTTP servers should include the same header fields in response to a `HEAD` request as they would in response to a `GET` request. Consequently, an LDP server should include the above header fields in response to a `HEAD` request as well.

The LDP recommends clients [Speicher 2015] to create resources via `POST` to a known LDP Container. LDP servers are required to respond to a successful with status code `201 Created` and to include the `Location` response header [Fielding 2014c] with the URL of the newly created resource.

In summary of our discussion, it is worth to note that the LDP promotes data-driven interaction and it enforces the HATEOAS constraint (see Section 2.1.1) via multiple normative requirements. We consider the LDP could be a suitable standard-compliant candidate for interaction in the envisioned IoT ecosystem.

2.4 The Web of Things

In this section, we discuss the integration of physical devices into a Web of Things (WoT), the motivation behind the WoT, how it relates to the Internet of Things (IoT), and recent developments in this area.

2.4.1 The need for an application architecture

Many definitions have been given for the IoT [Atzori 2010], however the underlying vision remains the same: the IoT integrates devices and everyday objects in a ubiquitous network. Nevertheless, while things become connected at the network layer, they remain isolated at the application layer. Given the envisioned diversity of connected things, it is unrealistic to imagine a common middleware for all things. To address this problem, some researchers turned to the Web as an integration platform for things [Kindberg 2002, Wilde 2007, Guinard 2010b]. The emerging Web of Things (WoT) is envisioned as an application architecture for the IoT.

The WoT vision is rapidly gaining ground. In June 2014, Dave Raggett, who had already coined the term *Ubiquitous Web* at W3C, has hosted the *W3C Workshop on the Web of Things*[13] together with Siemens. One of the outcomes of the workshop was the creation of the *W3C Web of Things Interest Group*[14]. Siemens has also recently created a WoT research group.[15] In October 2014, Google announced its Physical Web project[16], followed by a call for research proposals for an Open Web of Things in December 2014[17].

2.4.2 A resource-oriented architecture

Guinard et. al. [Guinard 2010b] proposed a resource-oriented architecture for the WoT. This proposal is elaborated as a layered architecture for the WoT in Guinard's Ph.D. dissertation [Guinard 2011a], structured around four layers:

1. *Accessibility* layer: deals with integrating things into the Web;

2. *Findability* layer: deals with searching for relevant services in the WoT;

3. *Sharing* layer: deals with managing access to things;

[13]http://www.w3.org/2014/02/wot/, Accessed: 08.11.2015

[14]http://www.w3.org/2014/09/wot-ig-charter.html, Accessed: 08.11.2015

[15]http://www.usa.siemens.com/en/about_us/research/web-of-things.htm, Accessed: 09.11.2015.

[16]http://techcrunch.com/2014/10/02/google-the-physical-web/, Accessed: 08.11.2015

[17]http://googleresearch.blogspot.fr/2014/12/call-for-research-proposals-to.html, Accessed: 08.11.2015

4. *Composition* layer: deals with integrating services across smart things, and thus supports composite WoT applications.

In this section, we mostly focus on the Accessibility layer, which we consider to be the essence of the WoT architecture. In Chapter 3, we discuss emerging paradigms and platforms for the WoT, which could be fitted along the remaining three layers.

Integrating things as first-class entities of the Web boils down to two main issues: designing RESTful things and placing them on the Web.

2.4.2.1 Designing RESTful things

In what follows, we discuss in further detail applying the *uniform interface constraint* (cf. Section 2.1.1) to design RESTful *things*. This discussion is based on [Guinard 2010b].

Identification of resources

In the context of the WoT, a *resource* is any component of an IoT device or application worth being uniquely identified [Guinard 2011a]. In compliance with the REST constraints, each resource is uniquely identified through a URI. For instance, the temperature of a thermostat might be identified through the following URI: `http://<DOMAIN>:<PORT>/devices/thermostats/<DEVICE_ID>/temperature`.

Resource representations

Guinard [Guinard 2011a] suggests that things should support at least HTML representations for human-to-machine interactions and JSON representations for machine-to-machine interactions. JSON was chosen over XML for being less verbose and thus better suited for resource-constrained devices.

Operations

Guinard [Guinard 2011a] suggests that a RESTful thing should support the HTTP verbs as basic operations, such as:

- `GET` for reading the temperature of the thermostat;
- `PUT` for writing the temperature of the thermostat;
- `POST` for creating new resources, such as a rule that increases the temperature to a given threshold after 6 a.m.;
- `DELETE` for deleting resources, such as the previous rule.

Not all operations will be available for all resources, and thus Guinard [Guinard 2011a] suggests to support HTTP `OPTIONS` for retrieving the list of available operations for a given resource.

HATEOAS

To satisfy the HATEOAS constraint (cf. Section 2.1.1), Guinard [Guinard 2011a] suggests that the representation of a resource should link to its parent, children and other related resources. Well-designed, hierarchical URIs help support this constraint as well. This recommendation encourages the development of a connected and crawlable WoT.

Implementing the above guidelines ensures that things are designed in a resource-oriented fashion, which is a first step towards their integration into the Web. The next step is to place things on the Web.

2.4.2.2 Web-enabling things

Two strategies are suggested for integrating things into the Web [Guinard 2010b]: either indirectly through the use of *smart gateways* acting as reverse proxies, or directly via embedded Web servers.

A *smart gateway* is essentially a Web server that shields resource-constrained devices and abstracts them through a RESTful API. Devices can use dedicated low-power protocols to talk to the smart gateway, such as Zigbee, Bluetooth Low Energy etc. Smart gateways can therefore provide an easy solution for integrating constrained devices into the Web. The disadvantage of this integration strategy, however, is that smart gateways have to be manually extended to support new protocols and things, and they are domain-specific intermediary components and thus do not conform to the end-to-end arguments of the Internet [Blumenthal 2001, Kovatsch 2015].

The direct integration of constrained devices into the Web via HTTP has already been proven to be feasible [Hui 2008, Dunkels 2009]. Going a step further, the standardization of the Constrained Application Protocol (CoAP) [Shelby 2014a] bridges the gap to directly integrate severely resource-constrained devices into the Web, that is devices with as little as 100 KiB of ROM and 10 KiB of RAM [Kovatsch 2015], also referred to as *Class 1 devices* [Bormann 2014].

Furthermore, CoAP provides features that are tailored for interaction in the IoT, such as publish/subscribe for *observable resources*, multicast communication, or alternative transports (e.g., SMS) [Shelby 2014a].

2.5 Summary

In this chapter, we discussed the principles underlying the Web architecture, we defined in further detail the problem of Web silos, and we discussed several facets of the Web that are relevant to our work. If the envisioned IoT ecosystem is to be a true global ecosystem, it is desirable to preserve the properties of the Web.

In Section 2.1, we discussed in detail one of the core tenets of the REST architectural style and we defined the "out-of-band information" problem. The latter provides greater insight into one of the main challenges we have to be address in order to free things from Web silos. That is, we must provide solutions to achieve *uniform interfaces* for heterogeneous APIs. To the best of our knowledge, there are no proposals that would enable the structured integration of non-uniform, non-hypermedia APIs into a global, hypermedia-driven environment.

In Section 2.2, we discussed the problem of Web silos in the context of the Social Web in search of solutions and technologies that could be useful to bring about the envisioned Social Web of Things (SWoT). Our investigation indicates that the technical solutions are already available. What is needed is an architectural model that would guide the development of the SWoT.

In Section 2.3, we touched on technologies that things could use to consume and produce content on the Web in a reliable fashion, and we found the LDP to be a suitable candidate for interaction in the envisioned IoT ecosystem. In Section 2.4, we discussed recent developments that enable the integration of resource-constrained devices into the Web. In the following chapter, we continue our discussion with an overview of emerging paradigms in the WoT.

Chapter 3

Emerging Paradigms in the Web of Things

Contents

In the previous chapter, we presented current approaches for the integration of physical devices into the Web, that is the so-called *Web of Things (WoT)*. In this chapter, we continue our investigation with a survey of abstractions, models and mechanisms proposed in WoT-related research. The purpose of this survey is to better identify and define the scope of the limitations that motivate our work.

In Section 3.1, we begin our discussion with an overview of current practices for integrating things into the Web. This discussion complements the one in the previous chapter and provides a better insight into how the WoT is currently being implemented. In this section, we also revisit the problem of Web silos (cf. Limitation 1) in the context of the WoT, and discuss approaches to achieve platform-level interoperability in the WoT. It is worth to note that *discoverability* in the WoT (cf. Limitation 2) is also related to the problem of Web silos: as long as things are confined to silos, they are not discoverable from the outside world.

In Section 3.2, we survey paradigms for interaction between people and things (cf Limitation 4). In Section 3.3, we survey approaches for integrating functionality across things in IoT mashups (cf. Limitation 3). In Section 3.4, we finalize our investigation by providing a survey of approaches that apply social concepts to the WoT and ubiquitous computing in general.

3.1 The Web of Things in Practice

In Section 2.4, we have presented a resource-oriented approach to integrate physical devices into the Web, which represents the widely held view in the WoT research community. In this section, we turn away from the academic field to present how the WoT is currently being implemented in the industry. The purpose of this incursion is to show that, in practice, the WoT is already evolving towards large silos of things, a fact that has already raised concerns in the research community [Blackstock 2014a]. This evolution further motivates our work, per Limitation 1.

In Section 3.1.1, we look at commercial IoT devices and analyze how they are being integrated into the Web. In Section 3.1.2, we discuss existing cloud-based IoT services and their approach to leverage the Web for the benefit of IoT developers. Following this discussions, in Section 3.1.3 we present current standardization efforts to enable interoperability in the WoT.

It is worth to note that we label as a WoT device or a WoT platform any device or platform that relies on interaction with other things via the Web.

3.1.1 WoT devices

Most present-day IoT consumer products are exposed to the Web via cloud services developed and maintained by their manufacturers. These cloud services usually provide a set of APIs, for instance, to read/write the state of a thing and subscribe to events. Some things implement the TCP/IP stack and can directly communicate with the cloud in a secure fashion, such as the WiFi enabled Nest thermostat[1], while non-IP devices rely on gateways to gain Internet connectivity, thus similar to the indirect integration approach presented in Section 2.4.2.2.

For instance, most wearable devices, such as wristbands or smart watches, that are usually resource-constrained and move around with the user are tethered to smartphones. The smartphone typically runs a dedicated application that handles communication to and from the cloud. Examples include fitness tracker series Jawbone UP[2] and Fitbit Flex[3]. Third-party systems can then interface with the cloud's RESTful APIs. For instance, retrieving a representation of the timezone of a Jawbone UP user may be performed via an HTTP `GET` request at the endpoint:
`https://jawbone.com/nudge/api/v.1.1/users/<user id>/timezone.`[4]

Non-IP devices that generally do not change their location often, such as smart light bulbs, are typically connected via bridges. Some bridges function similarly

[1]http://www.nest.com/thermostat/, Accessed: 12.11.2015.
[2]http://www.jawbone.com/up, Accessed: 12.11.2015.
[3]http://www.fitbit.com/fr/flex, Accessed: 12.11.2015.
[4]https://jawbone.com/up/developer/endpoints, Accessed: 12.11.2015.

to the smart gateways described in Section 2.4.2.2: they communicate with devices using low-power protocols and abstract them through a set of Web APIs. The bridge itself is usually not accessible to third-parties outside the user's private network, which means that applications run on the local network. Running applications locally presents the advantage of increased responsiveness, however the bridge may also be reflected by the manufacturer's cloud service to enable secure remote access. The Philips Hue[5] smart lighting system is one such example: a bridge communicates with the light bulbs via ZigBee Light Link[6] and exposes a set of RESTful APIs for reading/writing the states of lights and other abstract, non-physical resources, such as groups of light bulbs or schedules. The bridge provides unique URIs for all resources. For instance, a user may turn on light bulb `1` by sending a `PUT` request with the JSON payload `{ "on": true }` to the local endpoint `http://<bridge ip address>/api/<user id>/lights/1/state`.[7]

The main drawback of the above integration methods is that things are typically accessible via dedicated applications and heterogeneous APIs (see Section 2.1.2), which hinders interoperability in the WoT. It is also worth to note that most manufacturers of non-IP products do not currently provide APIs for direct access to their devices via low-power protocols, such as Bluetooth or ZigBee. The implication is that users will have to install dedicated components, such as smartphone applications or bridges, for each manufacturer. One of the few exceptions is Parrot, who also provide a set of Bluetooth APIs for their connected flower pots.[8]

3.1.2 WoT platforms

To address the above heterogeneity problem, *home hubs* abstract heterogeneous household appliances and personal devices behind a single application, which is used by people to manage things via the home hub, and a single API, which is used by developers to build applications. This abstraction simplifies application development for home automation scenarios, which in turn provides end-users with a wider variety of applications or, if so they choose, a single application to control all their devices. SmartThings[9] is one well known home hub manufacturer. SmartThings provides a taxonomy of capabilities for supported devices. Using this taxonomy, developers can create applications in the form of Groovy[10] scripts that run in a sandboxed environment. Users explicitly authorize access to a list of devices at installation time. SmartThings applications can also expose a set of Web APIs accessible to external systems via the SmartThings cloud. Developers have the flexibility to design their own endpoints and route them to appropriate handlers within the context of their

[5]http://www.meethue.com/, Accessed: 12.11.2015.
[6]http://www.zigbee.org/, Accessed: 12.11.2015.
[7]http://www.developers.meethue.com/documentation/core-concepts/, Accessed: 12.11.2015.
[8]http://flowerpowerdev.parrot.com/projects/flower-power-bluetooth-apis/, Accessed: 20.11.2015.
[9]http://www.smarthings.com, Accessed: 12.11.2015.
[10]http://groovy.codehaus.org, Accessed: 12.11.2015.

application. This feature allows greater flexibility for developers to extend the home hub platform, however it also increases heterogeneity in the WoT without standard guidelines for designing the APIs.

Home hubs are platforms that target household appliances and personal devices. In a more generic definition, a *WoT platform* can be defined as a system that provides "a repository for things (data and metadata) and a set of APIs for accessing and using things" [Blackstock 2014a]. Following this definition, Jawbone, Fitbit and Philips are IoT companies that have developed their own WoT platforms for the products they offer (see Section 3.1.1).

Xively[11], Evrythng[12] and ThingSpeak[13] are part of a new wave of IoT companies that focus on providing infrastructure as a service to developers and others willing to outsource the effort of developing and maintaining their own WoT platform. Other companies, such as ThingWorx[14] and AirVantage[15], offer end-to-end solutions for industrial IoT. In order to accommodate heterogeneous physical devices, each platform provides abstractions for things and the data they produce, such as channels or digital profiles. This results in an interoperability gain, within the boundaries of the WoT platform, and thus may foster the development of IoT applications for their customers. However, these WoT platforms do not usually interoperate with one another. Furthermore, each competing business naturally aims at dominating its IoT market segment and thus to create the larger "island of devices" [Blackstock 2013].

Following our discussion thus far, it appears that the WoT evolves towards a platform-centric model [Blackstock 2014a]. Existing WoT platforms are heterogeneous: they use different models, representations, authentication schemes. To achieve the vision of a global ubiquitous network of things and services, it is desirable to avoid replicating the problem of walled gardens in the WoT (cf. Section 2.2). To avoid WoT silos, it is necessary to achieve interoperability not only at the application layer, but also at the platform level.

In the following, we present an overview of the most prominent efforts to address interoperability issues in the IoT, and in particular in a platform-centric WoT.

3.1.3 Interoperability in the WoT

Interoperability requires standardization, and several efforts are underway towards standardizing the IoT/WoT. A comprehensive survey of IETF standardization in the IoT is available in [Ishaq 2013]. Most notably for the Web community is the work done by the IETF *Constrained RESTful Environments (CoRE)* working group[16], which aims at providing a framework for deploying resource-oriented applications on constrained IP networks. Important results of this working group include the

[11]http://www.xively.com, Accessed: 21.11.2015.
[12]http://www.evrythng.com, Accessed: 21.11.2015.
[13]http://www.thingspeak.com, Accessed: 21.11.2015.
[14]http://www.thingworx.com, Accessed: 21.11.2015.
[15]http://www.airvantage.net, Accessed: 21.11.2015.
[16]http://datatracker.ietf.org/wg/core/, Accessed: 21.11.2015.

Constrained Application Protocol (CoAP) [Shelby 2014b] and related initiatives (see Section 2.4.2.2).

A more comprehensive approach to IoT interoperability is proposed by the Internet of Things - Architecture (IoT-A) project[17], an impressive standardization effort in the context of the Seventh Framework Programme of the European Commission (FP7). IoT-A provides an architectural reference model for the IoT, which also addresses issues such as resource discovery and lookup or machine-to-machine (M2M) interfaces.

In 2014, the W3C has created a *WoT Interest Group*, which aims to provide, among others, use cases and requirements for the WoT, a standard high-level architecture, guidelines and best practices.[18] Other WoT-related initiatives for exposing sensors to the Web are undertaken by the *Open Geospatial Consortium*[19]. It is also worth to note that, in December 2014, Google launched a call for research proposals for an *Open Web of Things*.[20]

While the above organizations are working on comprehensive standards for IoT interoperability, some individual research groups are investigating ways in which WoT platform developers can already begin to address emerging interoperability issues. These initiatives may help standardization efforts to better define requirements and evaluate possible solutions for interoperability in the IoT. In what follows, we discuss in some detail one such approach proposed in the WoT research community [Blackstock 2014a]. To the best of our knowledge, this is currently the only proposal for a progressive strategy to achieve platform-level interoperability in the WoT.

3.1.3.1 A four stage path to hub-centric interoperability

Exposing things to the Web in a resource-oriented fashion already provides some degree of interoperability in the IoT and facilitates the development of service mashups across heterogeneous things (see Section 3.3). Still, developers are required to build adapters for the APIs of each individual thing or WoT platform. Blackstock et Lea proposed four stages towards achieving greater interoperability in a platform-centric WoT [Blackstock 2013]:

- *WoT Core*: IoT platforms expose things to the Web in a resource-oriented fashion, thus building a WoT;

- *WoT Model*: platforms agree on basic models, such as what things and data are managed; at this stage, developers may use a common set of APIs to retrieve a high-level catalogue of things contained on a given platform;

[17]http://www.iot-a.eu/, Accessed: 21.11.2015.

[18]http://www.w3.org/2014/09/wot-ig-charter.html, Accessed: 21.11.2015.

[19]http://www.ogcnetwork.net/IoT/, Accessed: 21.11.2015.

[20]http://googleresearch.blogspot.com/2014/12/call-for-research-proposals-to.html, Accessed: 08.11.2015.

- *WoT Hub*: at the next stage, platforms agree on implementation issues, such as security mechanisms, representation formats for accessing things in a generic fashion;
- *WoT Profiles*: to achieve a deeper integration, agreement on the semantics of things and the data they produce is necessary; at this final stage, platforms can directly link to and communicate with one another.

Platforms that achieve higher stages may thus be integrated more deeply with other interoperable platforms, facilitating the development of cross-platform applications.

It is worth to note that this strategy for platform-level interoperability can be used to achieve *uniform interfaces* for WoT platforms in a progressive manner, that is by allowing developers to balance platform design and implementation autonomy versus integration behind a standard interface. However, it does not enforce cross-platform interactions, which are necessary to enable *discoverability* (cf. Limitation 2).

3.1.3.2 HyperCat

To implement the strategy presented previously, Blackstock et Lea introduce HyperCat[21], a JSON-based [Bray 2014] hypermedia specification for representing and querying catalogues of resources [Blackstock 2014a].

Any HyperCat platform provides a top-level catalogue, which contains resources and may link to other catalogues. Catalogues and resources are denoted by URIs and described by lists of RDF-like triples, using a limited set of relations provided by HyperCat. The specifications also describe a set of operations, implemented as HTTP requests, to insert, update and delete catalogue items. Simple search queries may also be supported, in which case catalogues are required to advertise the operation. To address security issues, HyperCat specifies a simple authentication scheme. HyperCat was developed in the context of a UK government funded project focused on creating an open IoT ecosystem. Eight industry led sub-projects were tasked to develop domain-specific platforms for city transportation, smart homes, highways etc.

It is worth to note that the HyperCat specifications are quite similar to the Linked Data Platform (LDP) specifications [Speicher 2015] (cf. Section 2.3.3). Both HyperCat and LDP rely on two main abstractions: containers and resources, where containers are resources that may contain other resources. Resources can be described through triples. While LDP uses standard RDF serialization formats, such as Turtle [Beckett 2008] or JSON-LD [Sporny 2014], HyperCat uses JSON-based representations [Bray 2014]. Both specifications define operations, implemented as HTTP requests, for creating, updating or deleting resources in containers. LDP is a richer specification in terms of container types and operations. In addition, HyperCat specifies a simple security mechanism and a search operation. Therefore,

[21]http://wiki.1248.io/doku.php?id=hypercat, Accessed: 21.11.2015.

in our view, HyperCat can be defined by extending the LDP with WoT-specific vocabularies and operations.

Once again, we conclude that the LDP, which is a W3C recommendation, may be a suitable candidate for a standard-compliant interaction in the WoT (cf. discussion in Section 2.3.3).

3.2 Interacting with Physical Things

In the previous section, we surveyed approaches currently used in practice to integrate IoT devices into the WoT. Once physical devices and everyday objects are integrated into the Web as resources, people can directly access them using regular Web browsers. Alternatively, as noted in Section 3.1.1, many present-day IoT consumer products provide smartphone applications. Using dedicated Web/mobile applications for each individual device or task at hand, however, is not reasonable. An important challenge in ubiquitous computing is enabling people to manage, search and interact with large numbers of heterogeneous things [Randall 2003, Formo 2012, Takayama 2012], but also to coordinate and keep track of interactions between collaborative things [Brush 2011, Formo 2012, Mayer 2014a].

In this section, we present existing approaches to enable interaction between people and things. We survey such mechanisms in relation to Limitation 4, that is the problem of enabling people to *manage*, *interact with*, and *keep track of* large numbers of heterogeneous collaborative things.

Existing approaches for enabling people to interact with things can be classified in two categories [Mayer 2014a]: in Section 3.2.1, we discuss *local interaction*, in which people have physical access to things, and then in Section 3.2.2 we discuss *remote interaction*, for instance via search engines for the WoT.

3.2.1 Local interaction

IoT-related technologies embed sensing, actuation, processing and networking capabilities into everyday objects. Research in *human-computer interaction (HCI)* investigates how these technologies may be leveraged to embed user interfaces into people's daily lives in a seamless fashion, a research trend also referred to as *embedded interaction* [Kranz 2010a]. Other approaches investigate more explicit forms of interaction between people and things in their surrounding by means of *interaction devices*, such as smartphones [Derthick 2013] or smartglasses [Mayer 2014d], a research trend also known as *physical mobile interaction* [Rukzio 2006a, Broll 2009].

3.2.1.1 Embedded interaction

Mark Weiser envisioned a ubiquitous network that would weave itself into the fabric of everyday life [Weiser 1991]. With the proliferation of sensors, tagging technologies and tiny computing devices, this vision is rapidly becoming a reality. Off-the-shelf everyday objects already embed sensing capabilities to track usage and user activities in a seamless fashion, without requiring additional input from end-users, such as smart sneakers that monitor running or jumping[22] and mattress covers that detect movement and sleep cycles[23]. Other objects are augmented with capabilities to communicate with people non-intrusively [Streitz 2005, Rose 2014]. For instance, an umbrella may notify a user that it is going to rain by way of a discrete light [Rose 2014].

Going a step further, embedding intelligence in things can partially or completely eliminate interactions altogether. Such is the case of the Nest thermostat[24], which learns the temperature adjustment habits of its users in the first weeks of usage, or Parrot's autonomous flower pot featuring a self-contained irrigation system to provide one month of water autonomy [Lardinois 2015].

Prototypes for embedded interaction also extend to multiple objects that gather and share information. For instance, smart kitchen utensils may collaboratively track activities and usage in order to provide context-aware recommendations of recipes based on available ingredients [Langheinrich 2000], or variations of a recipe already being prepared [Kranz 2010a]. Smart-Its Friends [Holmquist 2001] provides a seamless interface that enables users to connect devices by holding and moving them together: Smart-Its devices use accelerometers to track movement data, which is then broadcasted to all other Smart-Its devices within listening range. If a device receives a movement pattern similar to its own most recent data, the two devices become "friends" and a dedicated connection is established. The connections established between devices may then be used in different applications.

In the above approaches, computers disappear: either they become small enough not to be observable, or they are no longer perceived as computing devices by users. Interaction is implicit and embedded into users' everyday tasks. These approaches, however, are focused on interaction with individual things and do not address managing or keeping track of large numbers of collaborative things.

3.2.1.2 Physical mobile interaction

In contrast with embedded interaction, *physical mobile interaction* [Rukzio 2006a, Broll 2009] is explicit and relies on mobile devices for selecting and interacting with things. Several techniques have been explored to select physical things by means of interaction devices [Rukzio 2006b]: by pointing the mobile device to the object (e.g., using a smartphone's camera to capture visual markers), by touching the object (e.g., using NFC tags) or scanning the environment for nearby things (e.g., via Bluetooth).

Several studies have been implemented to evaluate the usability of mobile user interfaces for appliances, as compared to traditional built-in interfaces. Some studies show that mobile interfaces are less efficient for everyday tasks than traditional

[22]http://nikeplus.nike.com/, Accessed: 02.12.2015.
[23]http://www.lunasleep.com/, Accessed: 02.12.2015.
[24]http://nest.com/, Accessed: 02.12.2015.

interfaces, and thus would not be the first choice of users [Roduner 2007]. More recent studies, however, focus on the dynamic and adaptive nature of mobile user interfaces, which may be useful for extending traditional interfaces. These studies reveal increasing support both from users [Hardy 2010, Mayer 2014a] and user experience designers [Derthick 2013].

Interaction devices may be particularly useful when entering new environments, or when interacting with multiple heterogeneous things. In [Mayer 2014d], wearable computers are used to select things by means of object recognition and to generate user interfaces on-the-fly, such as volume controllers for speakers. Mayer [Mayer 2014b] proposes augmented reality as a means to visualize message transmissions between things as they occur, to the aim of supporting people to keep track of interactions in smart environments. The main limitation of approaches based on object recognition, however, is that they provide accurate results only for relatively small predefined sets of things.

One prominent model for selecting and interacting with things by scanning the local environment relies on beacons that broadcast URIs in their surroundings. Users may then pick up the URIs using handheld devices, and bookmark or access them with any regular Web browser. This approach was first pioneered in HP Labs' Cooltown project [Kindberg 2002] by means of infrared beacons and PDAs. Bluetooth Low Energy beacons are now available at affordable prices. In October 2014, Google released an open-source project[25] to foster the development of URI beacons[26].

Similar to the embedded interaction mechanisms presented previously, most approaches for physical-mobile interaction are centered on interact with one individual thing at a time. The approach introduced by Mayer [Mayer 2014b] is targeting keeping track of interaction among heterogeneous things, however it applies only to real-time interaction in local environments. For instance, the user cannot keep track of his things in his smart home while at work, or investigate logs of their interactions later on.

3.2.2 Remote interaction

Previously, we discussed interaction mechanisms in which users have physical access to things. In this section, we generalize our discussion to selecting and interacting with physical things regardless of their location by means of discovery and look-up mechanisms, which are the predominant remote interaction mechanisms investigated in WoT research.

It is worth to note, however, that online social networks have also been proposed as mechanisms for remote interaction with things [Blackstock 2011, Formo 2012], which we consider to be complementary to the work presented in this section. We discuss social aspects in the WoT in Section 3.4.

[25]http://physical-web.org/, Accessed: 02.12.2015.
[26]http://google.github.io/uribeacon/, Accessed: 02.12.2015.

In some publications, WoT search is referred to as "entity discovery" [Romer 2010]. In other publications, "resource discovery" refers to crawling a thing to discover its properties and service interfaces, assuming that an entry point, such as the thing's URI, has already been found [Mayer 2011]. The discovered information is typically partially or completely indexed. WoT search is then composed of two steps: resource discovery and look-up, where the latter is concerned with retrieving relevant results, using the indices built in the discovery step, with respect to a given query. In the rest of this thesis, we generally follow the definitions given in the second approach.

In the next section, we present the problem of retrieving relevant physical things. We then look at search infrastructures prominent in the WoT context.

3.2.2.1 Discovery and look-up in the WoT

The problem of search in the WoT can be defined as "finding real-world entities with a given dynamic state" [Romer 2010]. The dynamic and contextual nature of information is central to the WoT and requires real-time search, which is a fundamental difference from indexing static Web documents. Furthermore, at the moment of writing this thesis, there is no standard way of describing smart things and the services they provide, and thus search engines for the WoT need to be extensible in order to adapt to new representation formats and resource description models [Mayer 2012]. Lastly, WoT search engines should provide results that are useful not only to humans, but also to software clients, for instance to look up services for automatic composition [Mayer 2014a]. To sum up, search engines for the WoT should be real-time, efficient, highly scalable, extensible and user-friendly, both to humans and machines.

3.2.2.2 Search infrastructures for the WoT

A survey of search engines for the IoT is presented in [Romer 2010]. Many of these search engines either retrieve (pseudo-)static content or raise efficiency concerns. Consequently, the authors propose Dyser [Ostermaier 2010], a real-time search engine for the WoT. Dyser is based on the assumption that enough sensors, such as the ones focused around human behaviors, provide predictable inputs. Dyser uses statistical models to rank indexed entities and sensors based on the likelihood of being in the state described by a given search query, for instance rooms that are empty. Data is pulled from sensors in descending order of computed probabilities until a number of hits is reached. Results are then ranked according to their relevance to the submitted query.

Dyser is focused on real-time, efficient and scalable search for the WoT. Another prominent discovery and look-up infrastructure for the WoT, which is to some extent complementary to Dyser, is proposed by Mayer et al. [Mayer 2012, Mayer 2014a]. This system focuses on extensibility, high scalability and user-friendliness: it discovers things, extracts metadata about their capabilities and makes them searchable for clients. It is a distributed infrastructure with nodes organized hierarchically. The motivation behind this design choice is to partition the search space by exploiting the locality of things, based on the assumption that physical things interact much more frequently with other devices in their immediate environment. To support high scalability, nodes communicate only with direct neighbors. For instance, a search would first be performed in a room, then expand to the entire floor, building etc. Clients, however, may also explicitly specify the scope of a search. One of the central features of the system is its high extensibility [Mayer 2011]: clients may add new mappings from a non-standard representation to an internal model, which is based on the Smart Thing Metadata model presented in [Guinard 2011a]. Furthermore, the internal representation model may also be extended through reflection, for instance if a mapping considers properties not supported by the internal model.

The problem of searching the WoT is complementary to our research objectives. It is worth to note that WoT search engines could benefit from enhanced *discoverability* in the ecosystem (cf. Limitation 2).

3.3 Physical Mashups

In the previous section, we discussed mechanisms for enabling interaction between people and things. In this section, we discuss mechanism for enabling interaction among things. We perform this investigation in relation to Limitation 3, that is the problem of static IoT mashups that do not scale.

One of the motivations behind applying the REST architectural style (see Section 2.1.1) to provide a resource-oriented architecture for the WoT (see Section 2.4.2) is that once physical things are decoupled from one another, which facilitates mashing up their services, that is the so-called *physical mashups* [Wilde 2007, Guinard 2009]. Furthermore, mashups editors similar to Yahoo Pipes![27] would then enable tech savvy users to program their smart environments. We discuss process-driven composition and physical mashup tools in Section 3.3.1. Following this discussion, we look at an approach for the automatic creation of physical mashups in Section 3.3.2.

3.3.1 Process-driven composition

As noted in Section 3.1, many IoT consumer products already provide Web APIs, while a new wave of WoT hubs provide services to integrate into the Web more traditional consumer products and industrial solutions. Once physical devices and everyday objects are exposed to the Web through RESTful APIs, developers may directly integrate functionality across heterogeneous things to create physical mashups. Going a step further, several tools have already been developed to facilitate the creation of physical mashups even for non-technical users. These tools typically leave the heavy lifting of data integration to developers and offer end users higher level abstractions, such as node templates or building-blocks, that may be easily integrated into mashups that run in the cloud.

[27]https://pipes.yahoo.com, Accessed: 03.12.2015.

One such Web platform that has been gaining wide popularity lately is IFTTT[28]. IFTTT enables users to mashup any of its supported services through simple rules of the form *if this then that*. IFTTT already integrates with over 160 services, which include SmartThings, Nest, Philips Hue, Jawbone, Fitbit, Android and iOS devices. For instance, users can build simple automations such as turning on the lights when they arrive home or receiving smartphone notifications if it rains the next day. Nevertheless, while IFTTT is very easy to use, it is also limited to mashups of two elementary services. Similar event-driven systems include Zapier[29] or the open-source Huginn[30].

More complex process-driven modeling may be done by means of physical mashup editors. These editors typically allow users to add and wire together building-blocks as abstractions for things and services. First proposals prominent in the context of the WoT were extending ClickScript[31], a JavaScript-based mashup editor, to include blocks featuring access to things [Guinard 2011a, Guinard 2011b]. More recent proposals include WoTKit [Blackstock 2012], glue.things [Kleinfeld 2014] and the highly popular Node-RED[32], an open-source platform launched in 2013 by IBM to foster the rapid prototyping of IoT applications. Unlike the mashup editors mentioned previously, Node-RED is a tool more accessible to developers and was not built to run mashups in the cloud. Node-RED is supported by a large community that contributes with node templates and mashups, referred to as *flows*[33].

The above editors are domain-independent. Other approaches offer even more user-friendly graphical abstractions for specific application domains, such as home automation. For instance, with homeBlox [Rietzler 2013] users create mashups by connecting different blocks, represented through icons, that denote human activities, household appliances, logical operators etc.

It is worth to note that all the process-driven platforms presented above are centralized. There are also initiatives to build distributed platforms, such as the prototype presented in [Blackstock 2014b]. Given the inherently parallel characteristic of process-driven models, the motivation is to take advantage of computing resources across devices and cloud-based services. The prototype extends Node-RED to support distributed flows by manually specifying devices for node execution.

The techniques to create physical mashups presented so far enable developers and tech savvy users to manually wire the WoT. This approach can be useful in several scenarios, from small-sized home automation to industrial automation. However, connections created between things are static, which means that the constructed physical mashups cannot adapt to dynamic environments or evolving user requirements. Furthermore, manually wiring the IoT does not scale (cf. Limitation 3).

[28]https://www.ifttt.com/, Accessed: 03.12.2015.
[29]http://www.zapier.com/, Accessed: 03.12.2015.
[30]https://github.com/cantino/huginn/, Accessed: 03.12.2015.
[31]http://www.clickscript.ch
[32]http://www.nodered.org, Accessed: 03.12.2015.
[33]http://flows.nodered.org/14.11.2015

3.3.2 Goal-driven composition

In order to avoid the adaptability and scalability problems inherent with the manual development of static physical mashups, goal-driven composition has been applied in the context of the WoT in [Mayer 2014c]: users specify their goals, such as playing a specific song, and it is left to the underlying infrastructure to deal with the automatic composition of services. Users may specify goals using visual language tools, by choosing from a set of available options etc.

In this approach, things add metadata to their services using RESTdesc [Verborgh 2011], a language for creating functional descriptions for RESTful services. A RESTdesc description is expressed in Notation3 and consists of a set of preconditions, a set of postconditions and an HTTP request that achieves the transition between the two states. Given the user's goals and descriptions of available services, a backward chaining reasoner is used to find an applicable sequence of HTTP requests such that the final state is achieved. If such a sequence is found, it is applied to the environment. A WoT discovery and look-up mechanism is necessary for retrieving available services (see Section 3.2.2.2).

3.4 Social Aspects in the WoT

Human social networks provide an intuitive metaphor for thinking about people, how they relate and interact with one another. The idea that pervasive computing could benefit from social concepts is gaining momentum. An extensive discussion of previous work on the convergence of social networks and the WoT/IoT is provided in [Ortiz 2014]. This convergence is frequently referred to either as the *Social Web of Things* [Formo 2012, Zhang 2012] or the *Social Internet of Things* [Atzori 2012], two emerging terms.

In this section, we summarize different approaches that apply social concepts to ubiquitous computing. We begin with platforms and applications that have been inspired by online social networks. Most of these initiatives have spawned from WoT-related research. We then extend our discussion to more general definitions of *social things* in the broader context of the IoT.

3.4.1 Platforms and applications

A number of research projects have used social networks for sharing Web-enabled things. For instance, SenseShare [Schmid 2007] proposed to use Facebook as front end to their system, relying on the platform's social graph for sharing sensor data with friends. Guinard et al. [Guinard 2010a] went a step further by providing support for several social platforms and a central point of access control. The system would also crawl things to discover the resources and operations available for sharing. Paraimpu [Pintus 2012] is another social tool that allows users to share, discover, bookmark and compose things. Therefore, Paraimpu is essentially a socially-enhanced physical mashup platform (cf. Section 3.3.1). In Paraimpu, other social networks, such as Facebook or Twitter, may be used as sensors/actuators.

Kranz et al. [Kranz 2010b] were among the first to explore the use of an existing social network, in particular Twitter, as a sensor/actuator in technological networks, thus building *socio-technical networks*. The authors, however, do not discuss a general architecture for developing such systems. Twitter, unlike other well-known social platforms, does not restrict its users to people: a Twitter user may be anyone or anything. Consequently, Twitter inspired a number of other IoT projects as well. López-de-Armentia et al. [López-de Armentia 2014] use Twitter as a communication platform for collaborative eco-aware appliances that rely on prediction models to infer how they should operate. The exchange of energy consumption patterns with similar devices bootstraps new eco-aware things, thus avoiding a cold start problem. In other approaches, things are microblogging about a user's mundane activities, such as an elderly going out for a walk, to enhance human-to-human interactions [Nazzi 2011].

Thing Broker [Perez de Almeida 2013] was also inspired by Twitter, and is based on the authors' previous experience with Magic Broker 2 [Blackstock 2010] and WoTKit [Blackstock 2012]. Thing Broker uses two main abstractions, *things* and *events*, to build a Twitter-like communication model. Relationships between things are unidirectional, similar to the following/followed relationship. Any data produced by a thing is encapsulated in events that are perceived by all followers. In the industry, a commercial WoT hub self-described as "Twitter for social machines" is dweet.io[34].

The above platforms and applications make limited use of existing online social networks. Blacktock et al. [Blackstock 2011] explore a stronger integration between existing social networks and the WoT. One observation is that some social platforms, such as Facebook or the ones implementing OpenSocial (see Section 2.2), may be extended with plug in applications, thus providing familiar user interface containers for WoT applications. Unlike SenseShare, the authors emphasize the need for WoT applications not to rely on one given social platform, but rather to be open to multiple such platforms. The authors go further and lay out key issues that need to be addressed towards a stronger integration with social networks, such as: identity authentication, extending social network models to accommodate things, dealing with privacy, thing state integrity and timeliness, maintaining dynamic relationships between people, places and things. We address several of these issues in Part II.

3.4.2 Social things

It is worth to note that most of the social aspects present in the WoT-related research in the previous section are centered around the Social Web, as it currently presents itself by means of different social platforms. Approaches in the broader IoT community tend to look at social aspects in a more generic fashion.

One of the first projects to explore the relation of friendship between objects was Smart-Its [Holmquist 2001] (see Section 3.2.1.1): users may hold and move together

[34]http://www.dweet.io/, Accessed: 17.11.2015.

Smart-Its devices to create connections between them, which may then be used in different applications. In other work, Vazquez et al. define *social devices* as objects that use the Internet "in order to communicate, collaborate, use global knowledge to solve local problems and perform in exciting new ways" [Vazquez 2008]. These devices "talk" to each other, locally or globally, at a *semantic* level by adding structure to and interpreting information. In doing so, devices use embedded reasoners and download ontologies from the Internet. Going a step further, authors envision artifacts that may be attached to living entities, such as plants, to make them appear intelligent. In a prototype implementation, a smart plant is able to perceive and interpret data in its environment to determine if conditions are suitable. If not, the smart plant asks its owner to be moved to a more suitable place using a synthesized voice. The plant thus becomes a first-class entity in its environment: it is *proactive* and influences its surrounding environment to achieve goals. We discuss autonomy and proactivity in Chapter 4.

Kortuem et al. are also considering autonomy and propose an “alternative architectural model for the IoT as a loosely coupled decentralized system of smart objects – that is, autonomous physical/digital objects augmented with sensing, processing, and network capabilities” [Kortuem 2010]. In this view, the application logic is divided across multiple physical things that “sense, log, and interpret what’s occurring within themselves and the world, act on their own, intercommunicate with each other, and exchange information with people”. The authors lay out some of the questions to be addressed: “what is the right balance for the distribution of functionality between smart objects and the supporting infrastructure? How do we model and represent smart objects’ intelligence? What are appropriate programming models? And how can people make sense of and interact with smart physical objects?”.

Researchers at the Ericsson User Experience Lab have illustrated, in a blogpost from 2012, a vision for the IoT that is similar to the one of Kortuem et al. presented above.[35] In their investigation, Ericsson was searching for an efficient mechanism to interact with large numbers of products and services. The initial approach of creating a mashup editor, similar to the ones described in Section 3.3.1, did not scale, both in terms of the user interface itself, but also in terms of understandability, or the ways in which users were able to comprehend large physical mashups. Online social networks are proposed as an alternative uniform interface for managing heterogeneous things. Things dynamically interacting with one another and their owner using the underlying social relations. Once again, things exhibit autonomous and goal-driven behavior.

Human social networks are also the driving force of Atzori et al.’s vision for a *Social Internet of Things (SIoT)* [Atzori 2012]. In this vision, things establish and manage their connections autonomously, by following the semantics of a pre-defined set of basic relation types:

- *parental object relationship*: established among things from the same produc-

[35]http://www.ericsson.com/uxblog/2012/04/a-social-web-of-things/, Accessed: 16.11.2015.

tion batch;

- *co-location object relationship*: established among things used always in the same place (e.g. in a home, city); the authors note that while it might be unlikely for some co-located objects to cooperate with one another, these relations are nevertheless useful for building "short" links within the network;

- *co-work object relationship*: established when things collaborate in an IoT application;

- *ownership object relationship*: established among things that belong to the same user;

- *social object relationship*: established when things come in contact due to encounters between their owners.

Aside from the behavior dictated by these types of relations, things are also compliant to any other rules that may be imposed by their owners. Related research investigates the structural properties of the SIoT network [Asl 2013], search optimization strategies [Nitti 2014a], task allocation [Colistra 2014] and trustworthiness management [Nitti 2014b] in the SIoT. We find the work on these various problems valuable and complementary to our research objectives.

It is important to note, however, that in the SIoT vision, people networks and object networks are two worlds apart. While these worlds could meet in SIoT applications, thus at a higher level, as the SIoT is currently presented, people and things are not first-class entities of the same system, which is an important difference from our vision and research objectives. Furthermore, a set of social relationships is proposed, which is based on the authors' interpretation of results from social sciences in the context of IoT applications [Atzori 2012]. While we find this approach innovative, and it provides a promising starting set of object relationships, we are not convinced that relations between things should be limited to a pre-defined set of relations. Previous work in WoT-related research points out the lack of consensus for describing things and any data associated to them, which motivates the need for generic and extensible solutions (see Section 3.1.3 and Section 3.2.2).

3.5 Summary

In this chapter, we presented a survey of emerging abstractions and models in the WoT. We discussed multiple aspects that are relevant to better identify and define the scope of the limitations that motivate of our work (see Section 1.1):

In Section 3.1, we discussed currently used strategies for integrating things into the Web and we observed that the WoT is evolving towards a platform-centric architecture, which stresses the need for platform-level interoperability. There is significant effort put into standardizing the WoT/IoT by several prominent organizations, such as IETF and W3C. In addition, individual research groups already provide developers with tools to address interoperability issues. We consider that approaching interoperability from both angles is beneficial for ensuring a good balance between standardization versus innovation. We presented a progressive strategy for achieving uniform interfaces for heterogenous WoT platforms based on a mediated model of catalogues of things. These approach, however, does not emphasize cross-platform interaction. It is also wroth to note that the many similarities between the specification underlying this approach and the LDP [Speicher 2015] supports our observation that the LDP could be a suitable candidate for standard-compliant interaction in the WoT (see Section 2.3.3).

Our discussion on integrating IoT consumer products into the Web points out that many such products currently rely on dedicated applications for usage. In Section 3.2, we surveyed other mechanisms to enable interaction between people and things. Research in human-computer interaction investigates the seamless integration of user interfaces into people's everyday tasks. Remote interaction, in which people do not have physical access to things, is currently addressed by means of discovery and search infrastructures. Furthermore, mechanisms that would enable people to keep track of interactions among heterogenous things in smart environments is also a researched topic. Existing approaches, however, do not address keeping track of large numbers of heterogeneous things and independent of their location.

In Section 3.3, we surveyed approaches that integrate functionality across heterogeneous things. Most approaches focus on creating static physical mashups of things and services in a (semi-)manual fashion. These approaches, while useful in many scenarios, also present a number of limitations. They do not scale well to large numbers of things, and they cannot adapt to dynamic environments or evolving user requirements. From a user experience perspective, creating and managing large physical mashups is cumbersome.

In Section 3.4, we discussed approaches that apply social concepts to ubiquitous computing. In WoT-related research, the term *Social Web of Things* is generally limited to leveraging features offered by existing social platforms, such as authentication, sharing access to things, or using online social networks as interface containers for things. In the broader IoT community, approaches focus more on endowing things with social abilities, such that they can dynamically connect and talk to one another. It is worth to note that most of the approaches presented in Section 3.4.2 raise the need for autonomy and proactive behavior, however they either do not provide mechanisms for enhancing things with such abilities, or they are limited to hard-coding a set of predefined behaviors in things. In the following chapter, we discuss results from multi-agent research in search of appropriate models and technologies to address this issue.

Chapter 4

Autonomy, Sociability and Regulation

Contents

In the previous chapters, we focused our discussion on the limitations that motivate our work (see Section 1.1). One of the core tenets of our thesis is to endow things with *autonomy*, a concept that is central to *multi-agent systems (MAS)*.

MAS is a vast domain. Similar to Chapter 2, the purpose of this chapter is to explore results from this domain that could help us bring about the envisioned IoT ecosystem. In Section 4.1, we begin with a general discussion about *autonomous agents* and *MAS*. In Section 4.2, we discuss models and mechanisms that enable agents to interact with one another and with their environment, and to reason upon the social context in which they are situated. In Section 4.3, we discuss approaches to enable control over autonomous behavior.

4.1 Multi-Agent Systems

In what follows, we discuss the properties of agents and MAS in Section 4.1.1, and in Section 4.1.2 we present the various dimensions typically used to study issues addressed in multi-agent research.

4.1.1 Properties of agents and multi-agent systems

An *agent* is commonly defined as "a computer system, situated in some environment, that is capable of flexible autonomous action in order to meet its design objectives" [Jennings 1998]. *Autonomy* is central to this definition and refers to the agent's ability to operate on its own, without the need of direct intervention from people or other agents. An autonomous agent thus has control over its actions and internal state.

An agent is typically situated in an external context or *environment*. The agent may perceive its environment by means of *sensors* and perform actions to influence its environment by means of *actuators*. What differentiates an agent from other autonomous systems [Jennings 1998], such as a software daemon that also operates autonomously in some software environment, is the agent's *flexibility* in the pursuit of some *design objectives*. Flexibility implies that the agent is *responsive* by reacting to changes in the environment in a timely fashion, *pro-active* by exhibiting goal-driven behavior and taking the initiative when appropriate, and *social* by interacting with humans or other agents in order to achieve complex tasks that would otherwise overcome its own capabilities.

Following the above agent definition, a *multi-agent system (MAS)* is a system conceptualized in terms of agents situated in a shared environment that interact with one another to achieve their objectives [Jennings 1998, Zambonelli 2003]. The multi-agent paradigm is thus suitable for building open and distributed systems situated in dynamic and complex environments, and which may interact with or act on behalf of humans [Boissier 2013]. Such systems are characterized by an inherent distribution of control, data, expertise or resources [Wooldridge 1995, Jennings 1998]. Furthermore, openness implies that agents and their intentions are not known in advance, they may change at runtime and they may be heterogeneous [Jennings 1998, Sycara 1998]. Consequently, given their properties of decentralization, distribution, heterogeneity and openness, MAS research takes into consideration issues such as communication, trust, negotiation, coordination and regulation.

4.1.2 Modeling dimensions for multi-agent systems

Given the complexity of the issues addressed by multi-agent research, they have been generally studied on a number of specific dimensions [Demazeau 1995, Boissier 2013], namely the *agent*, *environment*, *interaction*, and *organisation* dimensions.

The *agent* dimension is concerned with providing agent architectures, such as the *Belief-Desire-Intention (BDI) model* for cognitive agents [Rao 1995], and agent programming languages, such as Jason [Bordini 2007] or 2APL [Dastani 2008] that enable the direct implementation of agents in terms of BDI concepts.

The *environment* dimension is concerned with providing architectures, frameworks and infrastructures for modeling and implementing shared environments in which agents can coexist and evolve [Weyns 2005].

The *interaction* dimension is concerned with enabling communication and interaction among agents, for instance via agent communication languages [Labrou 1999] and interaction protocols [Demazeau 1995].

The *organisation* dimension is concerned with providing models, languages and infrastructures for defining, implementing and monitoring agent organisations, which typically use different abstractions, such as norms, schemes or structures, to formally specify and achieve collective objectives [Argente 2013].

Programming paradigms have been proposed for each of the above dimensions, namely agent-oriented programming [Shoham 1993], environment-oriented programming [Ricci 2011], interaction-oriented programming [Huhns 2001] and organisation-oriented programming [Boissier 2007]. Several multi-agent languages and platforms are discussed in [Bordini 2005]. More recently, Boissier et al. [Boissier 2013] have proposed the *multi-agent oriented programming* paradigm, which integrates the agent, environment and organisation dimensions in the JaCaMo[1] meta-model and platform.

Given the numerous programming paradigms and technologies for the development of autonomous agents and MAS, we consider multi-agent technologies can be suitable candidates for programming *autonomous things* in the envisioned IoT ecosystem.

4.2 Sociability in Multi-Agent Systems

Once we endow things with autonomy, our aim is to enable them to interact, in a flexible fashion, with other things and with their human users. In this section, we discuss further the *environment* and *interaction* dimensions in MAS. In Section 4.2.1, we begin with a discussion on shared environments, which can be used to embed mechanisms for indirect communication and coordination. We then discuss more direct forms of interaction in Section 4.2.2. In Section 4.2.3 we move away from interactions and focus on *social reasoning*, which agents can use to decide, for instance, who they should interact with or what social structures they should join.

4.2.1 Agents and artifacts

Thorough discussions of various approaches to model environments in MAS are available in [Weyns 2005, Platon 2007]. In a *situated MAS*, that is a system of agents that perform actions in some external context (cf. Section 4.1.1), the environment is typically a first-class abstraction that handles several responsibilities [Weyns 2007]: it provides agents with access to resources and services, it can actively manage its own state (e.g. evaporation of digital pheromones produced by ant-like agents), it can serve agents as a shared memory or as a medium for indirect interaction and coordination.

Some agent programming languages, such as Jason or 2APL, simulate environments as single computational objects. This monolithic approach implies that envi-

[1]http://jacamo.sourceforge.net/, Accessed: 12.03.2015.

ronments are centralized and may not be easily extended at runtime. In a different approach, the *Agents and Artifacts (A&A)* [Omicini 2008, Ricci 2011] meta-model defines an environment as a dynamic set of entities, called *artifacts*, which represent resources or tools that agents may use to achieve their design objectives. An artifact is made accessible by means of *operations*, that is computational processes executed inside the artifact that may be triggered by an agent or another artifact, and *observable properties*, which are state variables that may be perceived by agents observing the artifact. Operations can change the state of the artifact, and thus the values of observable properties, or can trigger *observable events* or *signals* to notify all observing agents about an event that occurred inside the artifact, such as an alarm triggered by a clock artifact. The artifact may provide agents with a manual, that is a machine-readable document containing a description of the artifact's functionalities. Artifacts may be linked together and are grouped in *workspaces*, which introduces the notion of locality or otherwise relatedness in environments.

The A&A meta-model makes a clear separation between the proactive parts of a MAS that are designed to achieve some goals, namely agents, and the passive parts, namely artifacts, that provide the former with the resources and functionalities needed to achieve their goals. Environment processes are encapsulated in artifacts, and therefore this approach inherently supports distributed environments. Furthermore, agents may create, destroy or "learn" how to use artifacts at runtime, which allows for the creation of a more flexible runtime environment. The A&A meta-model has been implemented in the CArtAgO platform [Ricci 2007a].

It is worth to note that artifacts provide a mechanism for merging the agent, environment, interaction and organisation dimensions in a flexible fashion [Boissier 2013]. An agent's external actions may be mapped to artifact operations and an artifact's observable properties to an agent's percepts, thus linking the agent and environment dimensions. Artifacts may also encapsulate mechanisms for indirect interaction and coordination [Omicini 2004].

4.2.2 Interaction

As discussed in the previous section, environments can embed mechanisms for indirect interaction among agents. In this section, we discuss forms of direct interaction among agents.

Research on multi-agent communication and interaction has been strongly influenced by the *speech act theory*, which we discuss in the following section. In Section 4.2.2.2, we discuss about *speech acts*, which provide primitives used by *agent communication languages* to enable a formal exchange of information between agents. In Section 4.2.2.3, we present how these exchanges can be structured by means of *interaction protocols*.

4.2.2.1 Speech act theory

Agent communication is typically based on speech act theory [Austin 1962, Searle 1969], which treats language as action. That is, similar to actions performed in the environment, an agent may attempt to change the state of the world by means of utterances. For instance, an agent may ask another agent to perform an action, a speech act identified by Searle as a *directive*. Other types of speech acts include *representatives* or *assertives*, which inform the hearer about something, *commisives*, which are promises made by the speaker to do something, *expressives*, which are used to express mental states or emotions, and *declaratives*, which change the reality with respect to the utterance.

Unlike actions, however, the domain of a speech act is limited to the mental states of the hearers [Bordini 2007]. In other words, the "direct" change that a speech act may bring in the world is in terms of the hearer's beliefs, desires or intentions. The hearer is then the one to interpret the utterance and possibly translate it to actions.

4.2.2.2 Agent communication languages

Speech act theory has been used as a theoretical framework for modeling agent communication [Labrou 1999]. Speech acts, also referred to as *performatives*, are typically used as primitives in *agent communication languages (ACLs)*. ACLs provide mechanisms for exchanging information among agents. Singh [Singh 1998] discusses two main approaches for designing ACLs. The first approach defines the meaning of speech acts in terms of mental states, such as beliefs, desires or intentions. Prominent mentalistic ACLs are KQML [Finin 1995, Labrou 1994] and FIPA-ACL[2]. The second approach defines the meaning of speech acts in terms of social commitments, such as an agent committing itself to the truth value of a statement made by means of an assertive speech act. One such commitment-based ACL is proposed in [Fornara 2002].

Singh [Singh 1998] argues that mentalistic ACLs focus on the private perspective of the sender and receiver involved in an interaction. However, in an open system composed of heterogeneous agents, an agent cannot make strong assumptions about the mental states of others [Singh 1998, Fornara 2002]. Furthermore, such mentalistic ACLs typically assume that agents are sincere and cooperative, and not for instance competitive. Therefore, this approach imposes constraints both on the design and the autonomy of agents. In contrast, commitment-based ACLs are focused on the public perspective of interaction in a MAS, which facilitates monitoring and testing for compliance. Therefore, this approach makes little assumptions on the agents' mental states or actions: social commitments are public, unambiguous and objective. This promotes heterogeneity and agent autonomy.

[2]http://www.fipa.org/specs/fipa00061/, Accessed: 02.12.2015.

4.2.2.3 Interaction protocols

ACLs enable the exchange of information among agents. Going a step further, *interaction protocols* specify how these exchanges take place between agents involved in an interaction, typically in the form of sequences of speech acts to be performed.

For instance, in the FIPA Contract Net Interaction Protocol[3], an extension of the contract net framework developed by Davis and Smith [Davis 1983], the initiating agent wishes to have a task performed by one or more participating agents. Furthermore, the initiator typically aims at optimizing a function that characterizes the task, such as a price or time to completion. To achieve this objective, the initiator first sends a *call for proposal* to a number of participants. Participants may accept the task and respond with proposals before a given deadline. The initiator evaluates the received proposals, and either accepts or rejects them. If a proposal is accepted, the participant informs the initiator of the result once the task is completed.

Standardizing communication by means of interaction protocols is particularly useful in the context of open systems of heterogeneous agents. The Foundation for Intelligent Physical Agents (FIPA) has standardized several interaction protocols, for instance for request-response interactions, publish-subscribe, auctions, or recruiting agents that are appropriate for given tasks.[4] An overview of the FIPA approach to standardization is available in [Poslad 2007].

4.2.3 Social reasoning

In the previous sections, we have discussed mechanisms for enabling interaction among agents, either directly or by means of shared environments. In this section, we move away from interactions and focus on another facet of sociability in MAS, namely *social reasoning*.

Sichman et al. [Sichman 1998] define *social reasoning* to be any type of reasoning mechanism that uses information about other agents to draw some inferences, and propose one such mechanism based on dependence relations [Castelfranchi 1992] among agents. An agent is dependent on another if the latter can help or prevent the former to achieve one of its goals [Sichman 2001]. In order to determine such relations, an agent needs to dynamically acquire and update information about the goals, resources, actions or plans of other agents, for instance by means of *introduction protocols* used when entering the agent society [Berthet 1992]. Using social reasoning and the computed dependence networks, agents can then evaluate at runtime if their goals are achievable or if their plans are feasible, they can dynamically form coalitions with other agents or revise their beliefs about other agents [Sichman 2001].

In other work, Carabelea et al. [Carabelea 2005] propose the use of *social power* to enable agents to reason about the constraints or gains when entering a group, such as an organisation or team. *Social dependence* and *social power* are to sides of

[3]http://www.fipa.org/specs/fipa00029/SC00029H.html, Accessed: 02.12.2015.

[4]http://www.fipa.org/repository/ips.php3, Accessed: 02.12.2015.

the same coin: an agent has social power over another that depends on the former.

4.3 Regulation in Multi-Agent Systems

Up to this point, we have explored how things could be enhanced with autonomy, how they could be enabled to interact with one another and to reason upon the social context in which they are situated. However, in an *open* ecosystem of autonomous things no prior assumptions can be made about the behavior of things, which raises the question of how can control be enabled over their autonomous behavior such that the overall ecosystem does not exhibit undesired behavior. That is to say, we are looking for mechanisms to provide autonomous, social things with a normative context to guide and regulate their interactions.

The study of *norms* in MAS is a research topic that has grown in importance over the past decades [Ossowski 2012]. In the following, we begin with a general discussion about *normative MAS* (see Section 4.3.1). Then, we focus our discussion on *social control mechanisms* (see Section 4.3.2) and *normative organisations* (see Section 4.3.3). The former are useful to enforce control in open and decentralized MAS, and the latter are useful to coordinate agents towards a desired collective behavior, such as achieving a complex task. Therefore we find these mechanisms to be of interest in the context of the envisioned open and decentralized IoT ecosystem of collaborative things. It is worth to note that the two approaches can be used in conjunction.

4.3.1 Norms in MAS

There exist several views on the definitions of *norms* and their role in MAS [Balke 2013]. Detailed explorations of normative concepts in MAS are available in [Ossowski 2012, Andrighetto 2013]. In what follows, we provide a rough overview of the concepts on which we base our discussion throughout the rest of this dissertation.

In the context of the SWoT we are interested, in particular, in norms that affect agent behavior. Some norms can be implemented in the environment or can be hard-coded into agents to *regiment* their actions. That is to say, actions that an agent should not be allowed to perform are simply not possible for the agent. However, not all norms can be implemented in such manner. Furthermore, regimentation severely restricts an agent's autonomy.

Another approach is to define norms that *regulate* the agents' behavior by describing what actions they are *obliged*, *prohibited* or *permitted* to perform. Such norms are also called *regulative norms* [Ossowski 2012] or *prescriptions* [Balke 2013] and they generally specify *who* does *what* in what *context* and as subject to what *deontic modality*. Regulative norms, therefore, affect agent behavior in an *indirect manner*. Consequently, in order to be effective, regulative norms require enforcement mechanisms that could, for instance, sanction norm violators.

It is important to note that agents can *decide* to conform or not to regulative norms, for instance, by balancing internal motivation versus external consequences.

Some norms can also be in conflict if, for instance, a norm obliges an agent to perform an action, whereas a different norm prohibits the agent to take the same action. Reasoning about norms, norm violations, conflicts and many other interesting norm-related subjects are studied by multi-agent research [Andrighetto 2013].

In the following sections, we continue our discussion with two mechanisms for enforcing norm-conformance, that is *social control* and *normative organisations*.

4.3.2 Social control

In a decentralized system of agents, in which there is no central authority, the task of enforcing norm-conformance belongs to the agents in the system. To this purpose, agents can use social constructs, such as *trust* and *reputation*, to discover trustworthy agents, to sanction norm violators, and to reason about committing violations.

Thorough discussions on computational models for trust and reputation are available in [Sabater 2005, Pinyol 2013, Ossowski 2012], and in the context of social networks in [Sherchan 2013]. Intuitively, *trust* refers to the relation between two agents, that is a *truster* and a *trustee*, whereas *reputation* is built based on experiences communicated by other agents. For instance, many of the existing online services, such as Amazon[5], eBay[6] or AirBnB[7] rely on opinions shared by members in communities built around the service. A detailed survey of trust and reputation in online services is available in [Jøsang 2007].

Mechanisms based on trust and reputation are susceptible to several vulnerabilities. For instance, malevolent agents, or groups of malevolent agents, can spread false information in the system to ruin one's reputation or to build their own reputation [Jøsang 2007]. Another well-recognized vulnerability is when agents with low reputation are able to change their identities and thus reset their reputation [Jøsang 2007]. Other vulnerabilities include discrimination [Jøsang 2009], for instance when an agent offers the same service with different quality levels to different agents, or reputation lag [Jøsang 2009], that is when an agent is able to take advantage of a period of time before its reputation is updated.

4.3.3 Normative organisations

Normative organisations can be applied to open and decentralized MAS to eliminate unwanted behavior or to achieve a coordinated behavior of the overall system. For instance, organisations can be used to coordinate teams of agents to achieve complex tasks. In contrast to *social control*, where control emerges from the independent actions of agents, organisations follow a top-down approach to regulating agent behavior. A state-of-the-art on multi-agent organisations is available in [Ossowski 2012].

An *organisation* is typically created by a *designer* using an *organisational model*, which provides a conceptual framework and a syntax that the designer can use to

[5]http://www.amazon.com/, Accessed: 04.12.2015.
[6]http://www.ebay.com, Accessed: 04.12.2015.
[7]http://www.airbnb.com/, Accessed: 04.12.2015.

create a specification of the organisation [Argente 2013]. Several organisational models have been proposed, such as AGR [Ferber 1998], ISLANDER [Esteva 2002], MOISE+ [Hubner 2007], and OMNI [Dignum 2005].

The organisation can then be implemented in a MAS by means of an *organisation management infrastructure* [Argente 2013], such as ORA4MAS [Hubner 2007], that interprets the specification and instantiates the organisation. Agents can use the infrastructure to join, leave, and participate in the organisation. Agents become aware of the organisation and can reason, for instance, before entering the organisation. The infrastructure can monitor norms in the organisation and apply sanctions to violators. It is worth to note that exposing such infrastructures via the Web could be beneficial to support interaction with heterogeneous agents in open MAS.

A detailed comparison of these models, and others, is available in [Coutinho 2005]. There is not yet a consensus on a unified model for describing organisations. Several dimensions are typically used to this purpose (see [Coutinho 2005]). Groups and roles seem to be common to all models, even though their semantic is not unique. In most cases, however, they are considered to be abstractions of a context for collective actions, respectively of status. Both are used to express norms or regulative behaviors.

4.4 Summary

In this chapter, we discussed models and technologies that could be useful to endow things with *autonomy*, *sociability*, and to make them *susceptible to regulation*. Multi-agent research provides a rich toolbox that could help achieve each of these three characteristics.

In Section 4.1, we discussed the properties of agents and multi-agent systems (MAS), we discussed the various modeling dimensions that can be used to address research topics in MAS, and we referenced several agent programming languages and multi-agent platforms that could be used to endow things with *autonomy* and build systems of autonomous things.

In Section 4.2, we discussed models and mechanisms for enabling agents to interact with one another and with their environment, and also to reason upon the social context in which they are situated, such that, for instance, they can decide whom to interact with. These models and mechanisms could be useful to endow things with *sociability*.

In Section 4.3, we discussed models for providing agents with a normative context that would guide their evolution and interactions, and which could thus be useful to enable control over the autonomous behavior of things.

Part II

Designing a Social Web of Things

Chapter 5

A Layered Architecture for the Social Web of Things

Contents

In Part I of this dissertation, we identified several limitations that hinder the development of the Internet of Things (IoT), and we have discussed current developments in the World Wide Web and multi-agent research that could prove beneficial in addressing these limitations.

In this chapter, we introduce our vision for an IoT ecosystem that transcends the identified limitations, which we call the *Social Web of Things (SWoT)*. We define a vision to be a destination accompanied by a sensible path to reach it. We illustrate our destination in Section 5.1 by means of several application scenarios constructed around the identified limitations. We discuss the principles that define and shape our approach in Section 5.2. In conformance with these principles, we propose a layered architecture for the envisioned IoT ecosystem in Section 5.3.

5.1 Application Scenarios

In what follows, we present four application scenarios. Each scenario has been designed to showcase what can be achieved by addressing one or more of the identified IoT limitations that motivate our work (see Section 1.1). The "Social TV" scenario in Section 5.1.1 showcases *discoverability* in the SWoT (see Limitation 2). The "Wake-up call" scenario in Section 5.1.2 showcases *flexible interaction* in the SWoT (see Limitation 3). The "Laundry room" scenario in Section 5.1.3 showcases *uniform remote interaction* with heterogeneous things (see Limitation 4). The "A welcoming home" scenario in Section 5.1.4 showcases *coordination* among autonomous things in the SWoT (see Limitation 3). In all scenario we assume that things are free from silos and able to autonomously participate in the SWoT (see Limitation 1).

We conclude this section with a discussion of the properties of the IoT ecosystem depicted in these scenarios.

5.1.1 Discoverability: The social TV

David has bought a social TV. During the installation process, David connects the TV to his STN Box[1], a WoT home hub that interconnects all of David's devices and services. What distinguishes the social TV from other similar products is that it can autonomously discover and interact with other social things to exchange information or provide complex functionality. For instance, the social TV can aggregate data from David's home appliances and display it in a dashboard, or it may summarize and display news from David's online social networks, such as Facebook or Twitter. These features are either implemented by the TV's manufacturer or provided via third-party applications.

David installs a third-party application on his TV to receive movie recommendations based on data aggregated from other social TVs owned by friends. The data can be movie ratings, movies watched recently, movies currently being watched, explicit recommendations made by friends etc. Once the application is installed, David's TV starts crawling the Social Web to discover all of David's friends and any social TVs they may own. The social TV can also expand its search to a depth of multiple levels in David's social graph, e.g. to friends of friends. Then, the social TV interacts with each of the discovered TVs to aggregate the data it needs for providing recommendations. The social TV is able to perform all of these tasks autonomously, while conforming to the terms imposed by any service it uses in the process. David's TV can also provide data to other social TVs, as long as the interaction complies with David's privacy policies.

In this scenario, David's TV exhibits several properties: it is *autonomous*, that is to say it can function with minimal intervention from its owner, it exhibits *goal-driven behavior*, such as determining and performing a course of action to provide movie recommendations to David, *it is aware of and conforms to regulations*, such as the ones imposed by terms of service or David's privacy policies, and it is *social*,

[1]STN is an acronym for *socio-technical network*, a concept that we define formally in Chapter 6.

in the sense that it is part of an open society in which it can interact with other social things *in a flexible manner*.

The ecosystem in which social things coexist in this scenario exhibits two properties that are central to the application for movie recommendations. First, the ecosystem is sustained by heterogeneous platforms (e..g, existing social platforms), however, David's social TV is able to operate seamlessly across all these platforms (cf. Limitation 1 in Section 1.1). Second, the ecosystem supports discoverability (cf. Limitation 2 in Section 1.1), which enables David's social TV to discover and interact with other social TVs in a flexible manner (cf. Limitation 3 in Section 1.1).

5.1.2 Flexible interaction: The wake-up call

It is 9:00 AM and David has a meeting scheduled in one hour. However, according to his social wristband and smart mattress cover, David is still asleep. His online social calendar implements a behavior that enables it to interact with other social things owned by David in order to wake him up. For instance, David's social wristband can try to wake him up at any time via vibration alarms, his social curtains can try to wake him up if there is light outside and the curtains are closed, his social lights can try to wake him up if the curtains are closed or if it is dark outside, and his social smartphone can try to wake him up at any time via sound alarms. David also has a preference for the order in which he should be woken up: he would choose at any time vibration alarms and natural light over artificial light or sound alarms.

David's social calendar first has to determine which of David's social things can wake him up. If David is away on a business trip, for instance, then it would make little sense to try to wake him up via his social curtains or social lights at home. In this particular case, however, given that David's mattress cover also supports the claim that he is asleep, it is very likely that David is sleeping at home. The outside light sensor indicates that the light level is well over 1000 lux (S.I.), and thus it is a sunny day[2]. The curtains are closed. After "talking" to David's social things, the social calendar determines that David's wristband, curtains, lights and smartphone could all try to wake him up. Per David's preferences, the social calendar asks David's wristband for a first attempt. The calendar relies, once again, on information provided by the wristband and mattress cover to determine if the attempt is successful or not. Should the attempt fail, the calendar escalates the alarm type used with each new attempt.

This application scenario stresses the ability of David's social things to interact *in a flexible manner* (cf. Limitation 3 in Section 1.1): heterogeneous social things are able to coordinate on-the-fly in order to determine which of them can wake up David, and do so in accordance with his preferences. Furthermore, the wake up feature should not break if David replaces his wristband or lightbulbs with products from different vendors. This scenario also highlights new abilities exhibited by David's social things: they can interpret and react to relevant information in their

[2]According to light levels provided by http://www.engineeringtoolbox.com/light-level-rooms-d_708.html, Accessed: 03.10.2015.

environment, and they are rational, that is to say they are able to make decisions and act upon them.

5.1.3 Remote interaction: The laundry room

Andrei is a Ph.D. candidate who lives in a student house in Saint-Étienne. The building has 6 floors, each floor with 20 studios and one laundry room. Each laundry room contains one washing machine and one dryer.[3] Andrei is usually caught up with work during the week, for which reason he typically takes care of his laundry during weekends. Unfortunately, so are most people living in the student house, and thus the laundromats typically get crowded on Saturdays and Sundays. Furthermore, due to the noise caused by the washing machines, which disturbs students living in adjacent rooms, the laundromats may be used only within specific hours. Occasionally, some machines may be out of service, or the payment units may fail or get jammed with coins. On other occasions, students forget or ignore the estimated time for washing or drying their clothes and do not return to retrieve them, thus blocking the machines without a way of being notified by others. All these factors sum up to a bad user experience. On a typical laundry day, Andrei has to make several trips to find a laundry room with a washing machine that is operational and available, either on his floor or on a different floor.

The administration has decided to replace all washing machines and dryers with a new generation of social machines that can interact with students via social platforms. It is worth to note that the machines can be heterogeneous: they can be produced by various vendors and they can be replaced over time. When a student registers at the administration, he indicates a preferred social platform that supports things as full-fledged users, such as Twitter[4]. The machines then autonomously connect with the student on the indicated platform such that they can receive and reply to messages. For instance, Andrei can now use Twitter to discover available machines or to book a time slot. To this purpose, Andrei posts a tweet, and the washing machines reply with their availability. The washing machines can also use social platforms to notify students, for instance, when the laundry is done or to remind them that they have yet to pick up their laundry.

In this application scenario, students use online social platforms as a *familiar* and *uniform mechanism* for *remote interaction* with *heterogeneous machines* (cf. Limitation 4 in Section 1.1).

5.1.4 Coordination: A welcoming home

David is leaving his office after a long day. David's car is connected to his STN Box at home and can access it from anywhere. When David gets in his car, the car notifies all his other social things at home that he is to arrive in about 30 minutes.

[3]As the name of our character might already hint, it is worth to note that this scenario is, in fact, modeled after a real situation encountered by this Ph.D. candidate.

[4]On Twitter, "users can be anyone or anything": https://dev.twitter.com/overview/api/users/, Accessed: 03.10.2015.

David's social things coordinate to prepare a warm welcome. His vacuum cleaning robot checks the last time it vacuumed the place and, if necessary, starts vacuuming. The thermostat, which is typically on energy saving mode, starts warming up the place to David's preferred ambient temperature. When the car notifies David's social things that they have just arrived, the sound system plays David's preferred ambient music and, depending on the outside light level, either the curtains open, or the lights turn on and any open curtains close (David prefers that whenever the lights are turned on in the evening, the curtains are closed). The social things keep logs on David's STN Box for all the actions they take. The logs are organized as human-readable threads of messages, similar to the ones encountered on social platforms, such that David can inspect the logs at any time and understand with ease the reasons behind changes in his home environment. Furthermore, the STN Box provides an interface that allows David to specify a desired state for his home environment, leaving it to his social things to coordinate and achieve the given state.

Similar to the application scenario presented in Section 5.1.2 ("The wake-up call"), this scenario emphasizes the ability of David's *heterogeneous* social things *to interact in a flexible manner* and *coordinate to achieve a common goal*, that is preparing the house for David's arrival (cf. Limitation 3 in Section 1.1). However, in this scenario the interaction is more complex, and the common goal can be decomposed in subgoals, such as vacuuming the house or setting the ambient temperature, that may be subject to temporal operators (e.g., goals can be achieved in parallel, in sequence). It is worth to note, once again, that David's social things are aware of his preferences, such as closing the curtains in the evening. This scenario also stresses the need for mechanisms that enable people to keep track of interactions in smart environments, such as David's home, but also mechanisms for managing heterogeneous things (cf. Limitation 4 in Section 1.1).

5.1.5 Discussion

The application scenarios presented in this section depict an *open* and *self-governed* IoT ecosystem that is composed of people and *social* things situated and interacting in a *global* environment that *spans across the physical-digital space*. The environment facilitates *discoverability* and *flexible interaction* between entities in the ecosystem. The envisioned IoT ecosystem is able to cope with the *heterogeneity* of the social things inhabiting the environment and of the platforms sustaining the environment. We further detail these properties in what follows.

David's social things are *autonomous* and *social*, and they are *first-class citizens* of an open society of people and social things in which they autonomously manage their interactions, coordinate in the pursuit of common goals, and are aware of norms applicable within the society, such as David's privacy policies. It is also worth to note that David's social things are aware of their surroundings and able to react to relevant events, such as David heading home, by making decisions and acting upon them.

Things' autonomy and social ability determine two important characteristics of the envisioned IoT ecosystem as a whole: it is an *open* ecosystem in which both people and social things can join or leave the ecosystem at any given moment, and once they join they can autonomously interact with other entities; it is a *self-governed* ecosystem, in the sense that social things operate with minimal human intervention, however, without causing security and privacy violations, or exhibiting any unwanted behavior that can be otherwise regulated.

People and social things coexist in an environment consisting of a *physical dimension* and a *digital dimension* that interweave and augment one another. The two dimensions interweave via sensors and actuators: IoT applications can use the former to collect input and interpret the physical world, for instance to determine if David is sleeping, and the latter to reflect back into the physical world, for instance to wake David up. The two dimensions can augment one another in various ways. On the one hand, for instance, services running in the digital space can manipulate David's home to provide a warm welcome after a long day at work. On the other hand, a vacuum cleaning robot, whose logic "lives" in the digital space, could solicit the physical intervention of a human for repairs. In a different example, David can meet new people in the physical world and reflect such events in the digital world via his online social platforms. David's social things can then use the newly created relations to augment their search in the digital space, such as David's social TV searching for other TVs owned by David's friends. In the same time, David can meet new people in the digital world via his online social platforms, which may augment David's relations in the physical world.

The environment is *global* in the sense that it is an Internet-scale system that transcends geographical and organizational boundaries. For instance, David's car can access and use his STN Box, i.e. a WoT home hub, regardless of its current location, and David's social TV can access and use any online social platform. It is worth to note that we consider non-social things, such as David's smart mattress cover or outside light sensor, as being part of the environment.

The environment facilitates *discoverability* in the sense that it enables people and social things to discover other entities and relations among entities in the ecosystem. David's social TV, for instance, is able to discover David's *friends* and any social TVs they *own*.

The environment facilitates *flexible interaction* between social entities in the sense that people and social things can use the environment to decouple their interaction, for instance, via brokers: Andrei and the washing machines use Twitter as a central broker for posting and replying to tweets, leaving it to Twitter to deliver the messages to his followers. It is worth to note that discoverability is also a means for achieving flexible interaction: David's social TV can crawl the ecosystem to discover other social TVs it can interact with.

We continue to refer to the application scenarios presented in this section throughout the rest of this chapter. It is worth to note that we implement these application scenarios in Chapter 9 in order to evaluate our contributions.

5.2 Principles

In this section, we distill the properties discussed in the previous section into a set of requirements for bringing about the envisioned IoT ecosystem, and propose a set of principles to address these requirements. In Section 5.2.1, we define the foundational principles on which we build our approach. We present the general design principles that shape our proposal in Section 5.2.2. The foundational principles define the underpinning of our proposal, and the general design principles guide the various choices we make along the way.

5.2.1 Foundational principles

We base our proposal for an architecture for the envisioned IoT ecosystem on four foundational principles: *conformity to the REST architectural style*, *social connectivity*, *autonomy*, and *regulation*. Per our discussion in Section 5.1.5, we apply the first two principles to create a *global* environment that *spans across the physical-digital space* and supports *discoverability*, and the last two principles to enable things as *first-class citizens* of the envisioned *self-governed* IoT ecosystem.

It is worth to note that the foundational principles we put forward in this section are the most fundamental statements that we consider to be true and on which we base our approach. We validate these statements and our approach in Part III of this dissertation.

5.2.1.1 Conformity to the REST architectural style

In the envisioned IoT ecosystem, people and social things are situated and interact in a *global* environment that is sustained by *heterogeneous platforms* and *spans across the physical-digital space*. A global environment emphasizes the need for scalability in terms of numbers of components sustaining the environment, interactions among those components, and interactions between the environment and its inhabitants. A global environment must also be able to cope with the evolvability of the overall ecosystem, that is to say the independent deployment and evolution of social things and environment components. Scalability and evolvability are two of the architectural properties, among others, that are emphasized by the REST architectural style (see Section 2.1.1 for more details). The WoT initiative is already proposing to apply REST in order to interconnect IoT devices and services at the application layer (see Section 2.4), therefore bridging the physical and digital worlds. We build on this initiative and formulate our first foundational principle as follows:

Foundational Principle 1 (Conformity to REST)**.** The REST architectural style provides a means to create a global environment for the envisioned IoT ecosystem.

A feature that is central to REST is having a uniform interface between architectural components. The uniform interface hides implementation details such that components are loosely coupled to one another, which in turn allows them to be deployed and to evolve independently. Evolvability is essential for the envisioned

IoT ecosystem, for instance, such that social things do not have to be manually configured against each platform in their environment, or such that social things do not break easily as platforms evolve. It is worth to note, however, per our discussion in Section 2.1.2, that existing Web platforms, which includes social platforms and WoT platforms such as the ones in our application scenarios, generally provide APIs that violate the uniform interface constraint (cf. Section 2.2.1 and Section 3.1.3). Therefore, if social things are to inhabit a global environment sustained by heterogeneous platforms, our approach has to provide solutions to integrate in this environment platforms with non-uniform APIs.

5.2.1.2 Social connectivity

The environment of the envisioned IoT ecosystem facilitates discoverability (see Section 5.1.5). In a global ecosystem expected to accommodate billions of people[5] and billions of things[6], discoverability implies that the environment must be able to interconnect the ecosystem in an *effective* and *flexible* manner.

REST already enforces general connectivity among resources via navigable relations (cf. uniform interface constraint in Section 2.1.1). For instance, this constraint enables discoverability on the Web via hyperlinks. We suggest that more *effective* connectivity can be achieved in the envisioned IoT ecosystem by having specialized relations, that is to say relations using standard relation types, between people and things, that is to say resources using standard resource types (cf. Section 2.3). Standard types enable machines to reliably interpret information in the environment in order to perform informed searches, which may be useful, for instance, to perform specialized searches or to provide real-time search results, an important challenge in the WoT (see Section 3.2.2.2). For example, David's social TV in Section 5.1.1 is able to reliably discover other *social TVs* that are *owned by* David's *friends*. In other words, while general connectivity is already enforced in a RESTful environment, specialized relations would serve as "information highways" in the ecosystem, leading to more effective connectivity. To achieve *flexible* connectivity, the environment must also facilitate the manipulation of these information highways.

Per our discussion in Section 2.2, it is worth to note that online social networks are already changing the way in which information is interconnected and disseminated on the Web. Furthermore, a basic functionality provided by most online social platforms to their users is to manage relations with people and entities in general, therefore providing *flexible* connectivity.

We propose that we can address the need for effective and flexible connectivity in the envisioned ecosystem by extending and applying the social network metaphor to the IoT, and formulate our second foundational principle as follows:

[5]For instance, as of June 30, 2015, Facebook reports 1.49 billion monthly active users (http://newsroom.fb.com/company-info/).

[6]Analysts expect more than 50 billion of connected devices by the end of 2020 [MacGillivray 2013].

Foundational Principle 2 (Social connectivity)**.** The social network metaphor provides a means to enhance discoverability in the envisioned IoT ecosystem.

It is also worth to note that social platforms provide a means to decouple interaction between their users by operating as central brokers. For instance, in Section 5.1.3, Andrei posts a "tweet" to signal his intention to do his laundry, leaving it to Twitter to route the message to washing machines following Andrei. Furthermore, per our discussion in Section 3.4.1, online social networks have been identified as good candidates to provide people with familiar user interfaces to manage and interact with heterogeneous IoT applications in a uniform manner.

5.2.1.3 Autonomy

A *self-governed* IoT ecosystem (see Section 5.1.5) emphasizes the need for things that can function with minimal human intervention. David's social things, for instance, can autonomously react to changes in their environment or manage their interactions with other social entities. Endowing things with autonomous behavior is central to the envisioned IoT ecosystem and a cornerstone of our approach. Per our discussion in Chapter 4, multi-agent research provides a vast amount of models and technologies for programming autonomous agents and systems of autonomous agents.

We formulate our third foundational principle as follows:

Foundational Principle 3 (Autonomy)**.** Autonomy is a means to enable things as first-class citizens of the envisioned IoT ecosystem.

It is worth to note that autonomy is a broad concept. We discuss in further detail how we apply this concept in our proposal in Section 5.3.

5.2.1.4 Regulation

In a *self-governed* ecosystem, the behavior of autonomous entities must be susceptible to regulation mechanisms. Furthermore, in an *open* ecosystem, no prior assumptions can be made about the behavior of autonomous entities, which stresses the need for enforcement mechanisms. Multi-agent research provides various means for enforcing regulation in systems of autonomous agents (see Section 4.3 for more details).

We formulate our fourth foundational principle as follows:

Foundational Principle 4 (Regulation)**.** Regulation is essential to create a self-governed IoT ecosystem.

We consider autonomy and regulation to be two facets of the same coin. Without regulation, autonomy is potentially dangerous. Without autonomy, regulation is not needed.

5.2.2 General design principles

The foundational principles presented in the previous section provide the underpinning of our approach. The general design principles that we present in this section guide the various choices we make to define our proposal. We rely on three design principles, namely *generality*, *separation of concerns*, and *interoperability*, which we discuss in what follows.

5.2.2.1 Generality

The envisioned IoT ecosystem must be able to cope with the heterogeneity of things and platforms. Heterogeneity mandates the use of general solutions to facilitate the integration of existing systems and to preserve the design and implementation autonomy of new systems and applications. Generality is a design principle that is central to our approach and to the successful development and adoption of the envisioned IoT ecosystem.

5.2.2.2 Separation of concerns

The envisioned IoT ecosystem must be extensible such that it can easily adapt and evolve over time, in particular in the rapidly changing landscape of the IoT (e.g., to benefit from emerging WoT protocols and standards). Furthermore, an approach based on general solutions must be extensible such that it can respond to domain- and application-specific requirements. Extensibility motivates the need for separating the concerns of the overall system into modules that can be easily exchanged or extended.

An important requirement for the successful adoption of the envisioned IoT ecosystem is to achieve a low entry-barrier for the development and use of applications. Given the complexity of the overall ecosystem, it is thus necessary to provide developers and users with abstractions that enable them to cope with the envisioned complexity. In the presented application scenarios, for instance, it is useful to abstract and separate the behavior of social things from their implementation such that developers can program heterogeneous things in a uniform manner.

5.2.2.3 Interoperability

A software system that is to be long-lived, global and open, in which components can be deployed and can evolve independently from one another, must rely on standards and knowledge in easily standardizable forms in order to ensure interoperability among its various components. This principle is best demonstrated by the REST architectural style and its most well-known implementation, the World Wide Web.

5.3 Layered Architecture

In this section, we reason up from the foundational principles defined in Section 5.2.1 and apply the general design principles presented in Section 5.2.2 to define an architecture for the IoT ecosystem described in Section 5.1.5.

Our proposed architecture is depicted in Figure 5.1. Per Foundational Principle 1 (conformity to REST), the proposed architecture conforms to the REST architectural style and we use the World Wide Web[7] as a primary means to implement and deploy the envisioned IoT ecosystem on a global scale (cf. *WoT* layer in Figure 5.1). For this reason, henceforth, we refer to this ecosystem as the *Social Web of Things (SWoT)*.[8] It is worth to note, however, that the proposed architecture does not depend on Web technologies or a particular protocol stack, which makes our architecture *future-proof*, that is to say unlikely to become obsolete in the future.

The SWoT architecture is structured along four layers, from bottom to top (cf. Figure 5.1): *Agency*, *Social*, *Normative*, and *Application*, where the *Social* and *Normative* layers are optional. The main motivation behind the proposed layering is the separation of concerns principle (cf. Section 5.2.2.2). We discuss and motivate each layer in the rest of this section.

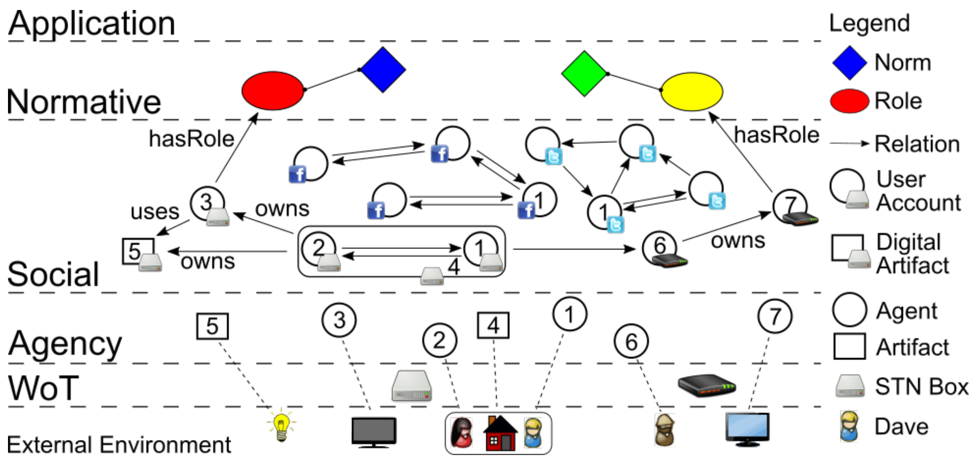


Figure 5.1: Our proposed layered architecture for the SWoT. We use numbers to identify *agents* and *artifacts* at the *Agency* layer with their associated *user accounts*, respectively *digital artifacts* at the *Social* layer. The icons representing *user accounts* and *digital artifacts* depict their hosting platforms in the lower right corner.

For illustrative purposes, Figure 5.1 depicts a partial representation of the "Social TV" scenario (see Section 5.1.1) and illustrates the various abstractions introduced by each layer. David and Sophia own an STN Box and a social TV. David also has a friend that owns an STN Box and a social TV. The "external environment" refers to all *entities* in the physical world, such as David and his family (i.e., a group).

[7]For instance, via its various facets discussed in Chapter 2.

[8]Our choice for the name of this ecosystem also highlights the central role played by the endowment of things with social behavior, which we discuss in Section 5.3.2.

Entities are abstracted at the *Agency* layer as *agents* and *artifacts*.

At the *Social* layer, *relations* among agents and artifacts are externalized in the environment and stored on various platforms (e.g., STN Boxes, Facebook, Twitter). These *explicitly represented* relations interconnect digital counterparts of agents and artifacts, which we call *digital artifacts*. The digital counterparts of agents are *user accounts*, which are a type of *digital artifact*. The relation between David and his friend is represented explicitly as a *cross-platform relation*.

At the *Normative* layer, externally defined *norms* can apply to various *roles* enacted by agents to regulate their autonomous behavior.

The concepts introduced in this section are defined formally in Chapter 6.

5.3.1 Agency layer

The *Agency* layer endows things with autonomy (per Foundational Principle 3) and abstracts entities in the SWoT as *agents* and *artifacts*, abstractions inspired by the Agents and Artifacts (A&A) meta-model (see Section 4.2.1). Our aim is to benefit from the vast amount of models and technologies provided by multi-agent research (see Chapter 4).

5.3.1.1 Agents and artifacts

We introduce the agent and artifact abstractions to separate exhibited behaviors from the actual entities in order to provide a uniform means of conceiving of people and heterogeneous things. In the SWoT, a thing can be any non-human entity worth modeling: physical (e.g., a lightbulb or a book), digital (e.g., an e-mail) or abstract (e.g., a research group).

The ecosystem is then composed of agents situated and interacting in a global environment, where the environment is represented as a dynamic set of artifacts, possibly organized in subsets, that agents can use to perceive and act on the physical and digital worlds, possibly by composing multiple artifacts to achieve new functionality (cf. A&A meta-model in Section 4.2.1). Therefore, in addition to separating behaviors from their implementations, the agent and artifact abstractions are also useful to separate the logic of agents from logic embedded in the environment, and to model the environment in a modular manner such that it can easily evolve, for instance via creating or deleting artifacts.

We conceive of people and things designed to inhabit the environment as first-class citizens of the SWoT and model them as agents. A defining characteristic of first-class citizens is their ability to *autonomously interact* with other entities in the SWoT. We conceive of things designed to be part of the environment as second-class citizens of the SWoT and model them as artifacts. Therefore, things can be modeled as either agents or artifacts, depending on their design purpose.

It is worth to note that in the A&A meta-model agents and artifacts are defined as first-class entities (cf. Section 4.2.1). In the SWoT, however, we define the dichotomy between first- and second-class citizens not from a software engineering perspective, but rather from the perspective of an external observer and based on the behavior exhibited by entities in the SWoT. The behavior exhibited by artifacts is typically much more limited than the one of agents inhabiting the environment, and thus we treat artifacts as second-class citizens of the SWoT. For example, the social TVs in Section 5.1.1 can navigate the environment to discover one another, and they can use the environment to persist and exchange messages, such as movie ratings via a platform designed for this purpose. We conceive of the social TVs as first-class citizens, and of the messages persisted in the environment, which are digital things modeled as artifacts, as second-class citizens.

5.3.1.2 Artifact-oriented interfaces

We have defined the environment as a dynamic set of artifacts. Some artifacts are non-persistent and, for instance, exist only within the running context of one or more agents, while other artifacts are persisted in the environment by means of platforms. Per our discussion in Section 5.2.1.1 (conformity to REST), it is central to the successful development of the SWoT to integrate in the environment heterogeneous platforms, such as existing social and WoT platforms. To address this issue, we have to provide solutions to achieve uniform interfaces for the APIs of heterogeneous platforms.

Our proposal is to abstract heterogeneous APIs by means of dynamic sets of artifacts, such as user accounts or messages, to create artifact-oriented interfaces. The interfaces resulting from the abstraction process have to be hypermedia-driven in order to conform to the uniform interface constraint (cf. Section 2.1.1). In other words, we abstract heterogeneous platforms by a layer of artifacts interconnected by typed relations. We discuss our approach in detail in Chapter 7.

What is achieved is a loose coupling between software clients and heterogeneous platforms in the ecosystem, where a software client can be a browser (i.e., a proxy for people), a software agent, or a multi-agent middleware (i.e., a proxy for one or more agents). Clients still need prior knowledge in order to reliably interpret and manipulate artifacts. We mitigate this aspect by applying the three design principles presented in Section 5.2.2 to decouple clients from platforms. We apply the generality and separation of concerns principles to define general artifact-oriented interfaces for specific domains, such as social platforms, that can be easily extended to satisfy subdomain- and application-specific requirements, and we apply the interoperability principle such that any domain-specific knowledge required to reliably interpret and manipulate artifacts is provided in easily standardizable forms, such as vocabularies or implementation guidelines (see Chapter 7 for further clarifications). We introduce one such general interface for social platforms when we discuss the *Social* layer in Section 5.3.2.2.

5.3.2 Social layer

The *Social* layer applies the social connectivity principle (per Foundational Principle 2) to externalize the relations among agents and artifacts into the environment. Externalized relations use standard types (e.g., friendship, ownership, provenance, colocation) such that they can be reliably interpreted and manipulated by both people and machines. This layer is optional. The separation of concerns it introduces simplifies the business logic of SWoT applications, enhances discoverability and enables flexible interaction in the SWoT.

5.3.2.1 A world-wide socio-technical graph

We refer to structures of agents and artifacts interconnected by means of typed relations as *socio-technical graphs (STGs).* STGs can be persisted in the environment, and possibly distributed across multiple platforms. We conceive of all STGs in the SWoT to form a world-wide, but not necessarily connected, socio-technical graph.

For example, in Figure 5.1 we illustrate these abstractions for the scenario in Section 5.1.1 ("The social TV"). David, his family, and their things are interconnected in an STG representing David's household, which is hosted on David's STN Box. Furthermore, David's STN Box is an open platform, and it allows David to add relations to agents in STGs hosted on other platforms, such as friends having their own household STGs and STN Boxes (cf. Figure 5.1). All household STGs are thus interconnected into a single STG distributed across multiples STN Boxes.

In addition, David also uses several social platforms, such as Facebook or Twitter, and each social platform hosts its own STG (cf. Figure 5.1). However, these platforms are closed and their STGs are disconnected from the household STGs. The problem, then, is that David's social TV cannot autonomously discover these STGs starting only from David's household STG.

The above example emphasizes the need for means to enable connectivity across disconnected STGs in order to enhance discoverability in the SWoT. It is worth to note that connectivity in the SWoT is closely related to the uniform interface constraint (cf. Section 2.1.1). We discuss this issue in further detail in Chapter 7.

5.3.2.2 Social artifacts

Following our previous example, the world-wide STG can be distributed across multiple heterogeneous and possibly closed platforms. Per our discussion in Section 5.3.1 on artifact-oriented interfaces, we model the APIs of heterogeneous platforms in terms of artifacts, and define five types of *social artifacts* that agents can use to access and manipulate the world-wide STG: *user accounts*, *digital groups*, *digital places*, *digital messages*, and *SWoT profiles.* We use the first four to provide general artifact-oriented interfaces for social and socio-technical platforms, and the latter to enhance connectivity in the SWoT. We motivate our choice for these particular types of social artifacts and provide formal definitions in Chapter 6.

User accounts represent digital counterparts of agents, and a software client acting via a user account is assumed to have been delegated by and acting for the user account's holder. Agents may hold multiple user accounts on various platforms. A user account typically holds all the information associated to an agent that is hosted by a given platform, such as an agent's relations. Relations in the SWoT can be established between entities (e.g., similar to FOAF networks in Section 2.2.3), or any user accounts they may hold, as it is generally the case in existing online social networks. Agents can thus crawl the world-wide STG by retrieving user accounts and navigating the relations they provide.

Digital groups represent digital counterparts of groups of agents. In its simplest form, a group is a set of agents, however it can also be defined by more complex social structures, such as the ones typically modeled in *multi-agent organizations* (see Section 4.3.3) by means of roles, relations among roles, subgroups, or power structures.

Digital messages represent digital counterparts of messages, the latter being interpreted as an abstract entity. For instance, a spoken message in the physical world can be reified and disseminated in multiple forms in the digital world, such as via an audio recording or reproduced as text.

Digital places represent digital counterparts of physical places and are necessary for anchoring STNs in the physical world, for instance, to attach a location to a message created by an agent.

SWoT profiles aggregate meta-information about agents in the SWoT, such as the user accounts they hold, and thus provide a means to construct an agent's distributed profile and increase connectivity in the SWoT. An agent typically has a single SWoT profile. For instance, David's social TV can use David's SWoT profile to autonomously discover all of his user accounts on various platforms. David's friends can also advertise their SWoT profiles on Facebook or Twitter in various ways (e.g., via their personal website), which would enable David's TV and other social things to discover their STN Boxes and interact with the things they own.

5.3.2.3 Flexible interaction

The social connectivity principle can be applied to interconnect entities in the SWoT via static or human-managed relations. However, given that agents are autonomous, they can also autonomously use and manipulate relations with other entities, which leads to flexible interactions. For instance, when David installs his newly acquired social TV (see Section 5.1.1), the TV can crawl David's home STN to discover other things owned by David and display them in a dashboard. Furthermore, it can crawl David's STG to discover social TVs owned by David's friends. If the TV encounters things of interest while crawling, it can add them to its STG such that it can reach them more easily in the future. Flexible interaction is central to our vision for the SWoT and an important consequence of the autonomy and social connectivity foundational principles.

Furthermore, as noted in our discussion of the social connectivity principle (see

Section 5.2.1.2), social and socio-technical platforms can function as central brokers that decouple communication between their users, which further simplifies the business logic of SWoT applications and enhances the flexibility, observability and controllability of interactions. SWoT applications are simplified because they do not have to deal with concerns of data storage and message routing. Flexibility is improved because agents can use platforms for one-to-many and many-to-many interactions, leaving it to the platforms to route messages based on relations already established in the STG. For instance, members of the same group or agents interested in the same topic do not have to be directly connected to one another in the STG. Observability is improved because all interactions pass through a central broker, which can be useful, for instance, to display them to the things' owners and thus improve SWoT usability, or to monitor norms that regulate interactions. Controllability, a dual aspect of observability, is improved because the brokers can apply norms, such as user-defined privacy policies, to regiment interactions or apply sanctions.

5.3.3 Normative layer

The *Normative* layer is concerned with expressing, monitoring and enforcing norms. This layer is optional. Nevertheless, externally defined regulation is central to create a global and self-governed SWoT (per Foundational Principle 4). It is worth to note that regulation can also be achieved at the *Agency* layer by hard-coding rules into agents, and at the *Social* layer via social control mechanisms (see Section 4.3.2).

5.3.3.1 Norms

We conceive of norms as deontic statements intended to regulate the autonomous behavior of agents. At the *Normative* layer, agents become norm-aware in the sense that they can interpret and reason on externally defined representations of norms. Agents, however, are also autonomous and heterogeneous, and some might decide to conform to norms, while others might choose to violate norms, which motivates the need for enforcement mechanisms. Per our discussion in Section 4.3, multi-agent research provides several models that can be used to express, monitor and enforce norms.

For instance, David's social TV should be able to retrieve and reliably interpret a platform's rate limiting policies and terms of service[9] to navigate across the SWoT. Autonomous vehicles should be able to dynamically obtain and interpret the driving regulations that apply to their current country or region. For another example, David's social things should be able to reliably interpret privacy policies defined by David that affect the dissemination of information into the SWoT.

In the above examples, norms are used to restrict the actions of agents and interactions among them. Norms, however, can also be used to incentivize agents to

[9]It is worth to note, for instance, that Twitter provides API access to its terms of service as unstructured text and to retrieve the rate limit status of an application as structured data.

take action. The latter can be used by externally defined coordination mechanisms, for instance to coordinate David's social things to achieve common goals, such as preparing the house for David's arrival (see Section 5.1.4).

5.3.3.2 Normative artifacts

In the SWoT, externally defined norms and enforcement mechanisms are encapsulated in *normative artifacts*. Normative artifacts can be created and used by agents at run-time, for instance via a multi-agent middleware such as the one we use in Chapter 9, or they can be persisted in the environment and hosted on platforms, similar to the social artifacts introduced in Section 5.3.2.2. However, we recognize that defining a general, uniform interface for working with normative artifacts is a complex endeavor and we leave this as future work (see Chapter 10).

5.3.4 Application layer

The *Application* layer is concerned with the business logic of SWoT applications. We use the term "SWoT application" loosely to refer to any software for end-users, which may be developed in terms of agents and artifacts or by extending the behavior of existing ones. For an illustration of the latter, in the scenario in Section 5.1.1 ("The social TV"), David installs an application on his TV to extend its behavior such that it can crawl the SWoT and aggregate data to provide movie recommendations.

The added benefit of the various layers introduced in our architecture is that, given the proper tools, developers can focus their effort on programming behaviors at a high level of abstraction. We support this claim with the results presented in Chapter 9, where we implement the application scenarios presented in Section 5.1 using an extension of the JaCaMo multi-agent platform [Boissier 2013]. Another benefit of the proposed abstractions is that they support the development of extensible IoT applications. For instance, if David's social TV is programmed as a *practical BDI agent* [Bordini 2007], then extending its behavior could consist in providing it with a set of plans, which the TV might also be able to obtain autonomously from other agents or from a repository of plans. It is worth to note that dynamically extending the behavior of software agents at run-time is in accordance with the code-on-demand constraint defined by the REST architectural style (see Section 2.1.1).

It is also worth to note that the various means of achieving regulation provided by the previous three layers can be used in conjunction in SWoT applications. For instance, David could define privacy policies to regulate the information his social things are permitted to disseminate into the SWoT. The privacy policies are defined at the *Normative* layer. Agents that can interpret David's policies are then able to determine if a message is subject to a privacy violation, and they can enforce privacy preservation via hard-coded rules (i.e., at the *Agency* layer) aimed to stop and prevent violations, such as:

1. Do not disseminate messages that are subject to privacy violations such that the violations are not propagated.
2. If a message received from another agent is subject to a privacy violation, delete the message and enforce social control by informing others about the violation.

At the *Social* layer, social control mechanisms can be used to gradually isolate recurring violators from the SWoT, for instance by breaking relations and communication with the violators. This example is based on a previous project we have been involved in, the PrivaCIAS framework [Krupa 2012] for privacy preservation in online virtual communities, and we have used this approach to implement a privacy-aware, open and decentralized photo-sharing social network for smartphones [Ciortea 2012].

5.4 Summary

In this chapter, we addressed Research Question 1, that is *how to bring systems of autonomous things to the Web of Things*. To this purpose, we proposed an architecture for a global IoT ecosystem that enables autonomous things as its first-class citizens. Autonomy is central to our proposal and provides the means to transcend the IoT limitations identified in Part I.

In Section 5.1, we introduced several application scenarios to illustrate the envisioned IoT ecosystem and define its properties. In Section 5.2, we distilled these properties into requirements for bringing about the envisioned ecosystem and, to address these requirements, we formulated a set of foundational principles that provide the underpinning of our proposal. In other words, the foundational principles are the most fundamental statements we consider to be true and on which we base our approach. We reason up from these statements, guided by a well-defined set of design principles, to build our proposal in Section 5.3. Our investigation comes full circle in Part III: in Chapter 9, we validate our foundational principles and proposal by implementing the application scenarios introduced in this chapter.

A central feature of the envisioned IoT ecosystem is the flexible interaction between its first-class citizens, which we achieve by applying the autonomy and social connectivity principles to enable agents to autonomously manage their relations and interactions with other entities. In Section 5.3.2.2, we introduced informally a set of social artifacts that agents can use to access and manipulate the world-wide socio-technical graph. We formalize our discussion in Chapter 6. In Chapter 7, we discuss solutions to create uniform interfaces in the SWoT by mapping artifact-oriented interfaces, such as the ones defined via social artifacts, to the APIs of heterogeneous platforms.

Chapter 6

A Model for Socio-technical Networks

Contents

In the previous chapter, we introduced a layered architecture for the Social Web of Things (SWoT). The essence of our approach is to endow things with autonomy (per Foundational Principle 3 in Section 5.2.1) and apply the social network metaphor (per Foundational Principle 2 in Section 5.2.1) to create flexible networks of people and autonomous things. We refer to these networks as *socio-technical networks (STNs)*. An STN spans across the four layers of the architecture proposed in Section 5.3, and we conceive of STNs as the building blocks of the SWoT.

The purpose of this chapter is to provide a formal model for STNs. We introduce and define the elements of our model in a top-down manner, from the most generic to the most specific. In Section 6.1, we begin our presentation with an informal overview of the various *entities*, organized in multiple *modeling dimensions*, that we use to represent STNs. We formalize our discussion in the following sections. In Section 6.2, we define a *general* and *extensible* mathematical model for STNs. This model is *general* in the sense that it can be used to create representations of heterogeneous STNs, such as a family or an online virtual community, at a high level of abstraction. This model is *extensible* in the sense that it can be easily specialized to represent domain-specific STNs. In Section 6.3, for instance, we specialize our *STN model* to represent *digital STNs*, such as online social networks. The *digital STN model* presented in this section provides formal definitions for the *social artifacts* introduced previously in Section 5.3.2.2 and we use it in Chapter 7 as a mediated model for the integration of heterogenous STN platforms.

6.1 Modeling Dimensions

Per the separation of concerns principle (see Section 5.2.2), we model STNs on multiple dimensions, namely the *social*, *normative*, *spatial* and *digital* dimensions. The *social dimension* is concerned with modeling relations and interactions among agents (cf. *Social* layer in Figure 5.1). The *normative dimension* is concerned with modeling normative concepts used to regulate the behavior of agents (cf. *Normative* layer in Figure 5.1). The *spatial dimension* is concerned with anchoring STNs in the physical space (cf. *external environment* in Figure 5.1). The *digital dimension* is concerned with reifying STNs in the digital space.

In what follows, we first introduce several preliminary definitions, which we then use to discuss each of the above modeling dimensions.

6.1.1 Preliminary definitions

For illustrative purposes, Figure 6.1 depicts an STN that models David's household at a given point in time. The Doe household includes David Doe and Sophia Doe, who are spouses and form a family, and the house that they own and live in. We use and elaborate on this example throughout the rest of this chapter. We illustrate the digital dimension of the Does' home STN in Section 6.1.5.

6.1.1.1 Basic entities of STNs

The STN depicted in Figure 6.1 is composed of multiple interconnected *entities*. Entities of an STN can be anything worth modeling, such as persons (e.g., `David Doe` and `Sophia Doe`), groups (e.g., `The Doe family`), physical objects (e.g., `The Does' house`), or concepts (e.g., `Agent`). It is worth to note that, per Foundational Principle 1 (conformity to REST), all entities in the SWoT are uniquely identified (cf. uniform interface constraint in Section 2.1.1).

Definition 6.1.1 (Entity)**.** An *entity* is anything worth modeling within an STN. It is uniquely identified in the SWoT.

Entities in an STN are interrelated in a meaningful manner by means of *relations* of given *relation types*. For instance, in our illustration in Figure 6.1, `David Doe` is the *spouse of* `Sophia Doe`, and `The Doe family` *owns* a house, i.e. `Does' House`.

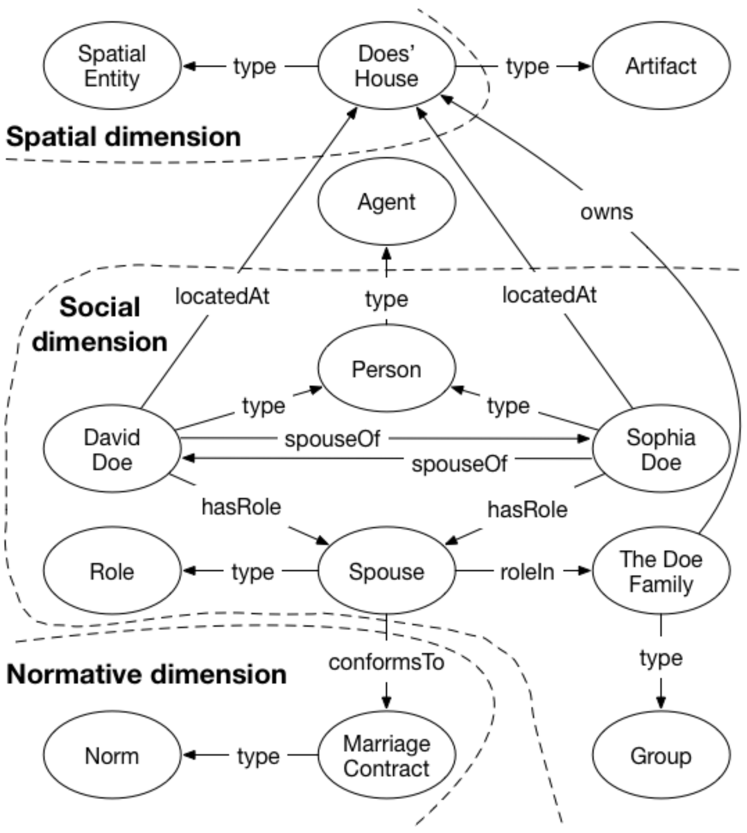


Figure 6.1: An illustration of the *social*, *normative* and *spatial* dimensions of the Doe household at a given point in time. At the moment depicted in this figure, both David and Sophia are at home.

Relation types, such as `spouseOf` or `owns`, enable people and machines to interpret and reason upon *relations.*

Definition 6.1.2 (Relation)**.** A *relation* is an *entity* that represents a unidirectional connection of a given *relation type* between two *entities.*

Definition 6.1.3 (Relation type)**.** A *relation type* is an *entity* that denotes a set of common characteristics shared by a category of *relations.*

As hinted by our illustration in Figure 6.1, entities in an STN can also be assigned one or more *entity types* via *typing relations.* `The Does' House`, for instance, is *typed as* an `Artifact` and a `Spatial Entity` via `type` relations.

Definition 6.1.4 (Entity type)**.** An *entity type* is an *entity* that denotes a set of common characteristics shared by a category of *entities.*

Definition 6.1.5 (Typing relation)**.** A *typing relation* is a *relation* that associates an *entity type* to an *entity.*

Per our discussion of the *Agency* layer (see Section 5.3.1), two specific types of entities in the SWoT are *agents* and *artifacts*. In Figure 6.1, for instance, `David Doe` and `Sophia Doe` are modeled as *agents*, and `The Does' House` is modeled as an *artifact*. It is worth to note that some entities are neither agents, nor artifacts, such as `The Doe family`, which represents a *group* of agents (see social dimension in Section 6.1.2).

Definition 6.1.6 (Agent)**.** An *agent* is an *entity* that is able to autonomously interact with other *entities* in the SWoT. An *agent* can be either a human or software entity.

In the application scenarios in Section 5.1, David owns multiple *social things*. Social things are agents in the SWoT that are *non-human*. They can embed the logic of a physical thing, such as David's social TV in Section 5.1.1, or a digital thing, such as David's social calendar in Section 5.1.2. It is worth to note that a social thing can be a composite application built on top of multiple things, that is an IoT mashup.

Definition 6.1.7 (Social thing)**.** A *social thing* is an *agent* that is non-human.

Agents can use *artifacts* to perceive and act on the physical and digital worlds. In our illustration, for instance, David and Sophia live in a house, that is `The Does' House` (cf. Figure 6.1). Agents can create mash-ups of artifacts in order to achieve composite functionality. For example, if the Does' Web-enabled lights and curtains are modeled as artifacts, their social TV should be able to create a mash-up such that if it dims the lights, the curtains close automatically. The social TV could then use this mash-up, for instance, when a movie starts playing.

Definition 6.1.8 (Artifact)**.** An *artifact* is an *entity* that *agents* can use in order to perceive and act on the physical and digital worlds.

Per our discussion in Section 5.3.1 (the *Agency* layer), it is worth to note that a thing can be modeled both as an agent and an artifact, depending on its design purpose. Furthermore, the same thing can be modeled as an agent in one STN and as an artifact in another.

6.1.1.2 Structure and dynamics of STNs

Having introduced all the basic entities of STNs, we define an STN as follows:

Definition 6.1.9 (Socio-technical network)**.** A *socio-technical network* is a dynamic and regulated system of *agents* and other *entities* interrelated in a meaningful manner via *relations*.

An STN is *dynamic* in the sense that it evolves over time. An STN is *regulated* in the sense that, per Foundational Principle 4 (regulation), the behavior of agents in the STN is regulated. Entities in an STN are *interrelated in a meaningful manner* in the sense that *relations* among *entities* are represented explicitly using standard *relation types* such that both people and machines can reliably interpret and reason upon them. Furthermore, *entities* in an STN are also assigned *entity types*.

In our discussion of the *Social* layer (see Section 5.3.2), we introduced *socio-technical graphs (STG)*. An STG represents a "snapshot" of an STN, that is to say all the entities in the STN and relations among them at a given point in time. The STG depicted in Figure 6.1, for instance, corresponds to a state of the illustrated STN in which both David and Sophia are at home. Therefore, an STG exists only within the context of an STN, and the evolution of an STN can be represented as a succession of STGs. The world-wide STG (see Section 5.3.2) is then composed of the interconnected STGs of all STNs in the SWoT, and each STG may evolve independently.

Definition 6.1.10 (Socio-technical graph)**.** A *socio-technical graph* is the state of a *socio-technical network* at a given point in time.

An STG can evolve by means of *operations* performed by *agents*. *Operations* that share common characteristics, such as common side effects on the STG on which they are applied, can be abstracted via *operation types*. For example, members of the Doe family can *acquire* things in two different ways: for their family, or for themselves. If David, for instance, acquires a social TV for his family, this *operation* can be reflected in the STG in Figure 6.1 by adding the social TV as a *social thing* that is *owned by* `The Doe Family`. If David acquires a social wristband for himself, the social wristband would be represented in the STG as a *social thing* that is *owned by* `David Doe`. The STG resulting from the application of an operation represents a new state of the STN.

Definition 6.1.11 (Operation)**.** An *operation* is an atomic activity performed by an *agent* on a *socio-technical graph*.

Definition 6.1.12 (Operation type)**.** An *operation type* is an *entity* that denotes a set of common characteristics shared by a category of *operations*.

Based on the primary definitions introduced thus far, in the following sections we define the *social*, *normative*, *spatial* and *digital* dimensions of an STN.

6.1.2 The social dimension

The social dimension of an STN is represented explicitly by means of *social relations*, *groups* and *messages*.

Social relations are *relations* established between two *agents*, such as the `spouseOf` relations between `David Doe` and `Sophia Doe` in Figure 6.1. Per the social connectivity principle (see Foundational Principle 2), their purpose is to enhance discoverability in the SWoT.

Definition 6.1.13 (Social relation)**.** A *social relation* is a *relation* established between two *agents*.

Groups provide a means to represent structures of multiple *agents*. In its simplest form, a group can represent a collection of agents, such as a collection of *social things* owned by the Doe family. However, groups can also represent more complex structures, for instance, by means of *roles* and *relations* among roles. For example, in Figure 6.1, `David Doe` and `Sophia Doe` are members of a group denoting `The Doe Family` in which they both enact the role `Spouse`. For another example, the roles in this group could also include `Parent` or `Child`, where the former could have, for instance, a relation of authority with the latter.

Definition 6.1.14 (Group)**.** A *group* is an *entity* that represents a collection of agents.

Definition 6.1.15 (Role)**.** A *role* is an *entity* that represents a status that can be enacted by an *agent* within the scope of a *group*.

Messages are pieces of information transmitted between agents. Messages can take various forms and be transmitted in various ways, such as spoken messages transmitted between David and Sophia or digital messages exchanged online.

Definition 6.1.16 (Message)**.** A *message* is an *entity* that represents a piece of information transmitted between *agents*.

The above examples are focused primarily on representing social relations and interactions in the physical world. It is worth to note, however, that these social relations and interactions can also be established and take place in the digital dimension, which we discuss in Section 6.1.5.

6.1.3 The normative dimension

The normative dimension of an STN is represented explicitly by means of *norms*. Norms can be attached, for instance, to roles or agents. For example, in our illustration in Figure 6.1, `David` and `Sophia` enact the role `Spouse` that conforms to a norm defining marriage obligations, such as determining the owner of `The Does' house` should the marriage break.

Definition 6.1.17 (Norm)**.** A *norm* is an *entity* that denotes an explicit specification of deontic statements (i.e., obligations, permissions, prohibitions) whose purpose is to regulate the behavior of *agents*.

It is worth to emphasize that agents are autonomous and can decide to violate or comply to norms.

6.1.4 The spatial dimension

The spatial dimension of an STN is represented explicitly by means of *spatial entities* and *localization relations*.

Spatial entities are entities in the SWoT that have spatial characteristics, such as a position or a surface. Their purpose is to provide a means to anchor STNs in the physical space. In Figure 6.1, for instance, `David` and `Sophia` are *located at* `The Does' House`, which is a `Spatial Entity`.

Definition 6.1.18 (Spatial characteristic)**.** A *spatial characteristic* is any feature that can serve as a reference point in the physical space.

It is worth to note that people have spatial characteristics, and thus are spatial entities. *David's wristband*, for instance, can be located on *David's wrist.*

Definition 6.1.19 (Spatial entity)**.** A *spatial entity* is an *entity* that has *spatial characteristics.*

We refer to spatial entities that are non-human as *places.*

Definition 6.1.20 (Place)**.** A *place* is a *spatial entity* that is non-human.

Localization relations provide a means to relate *agents* and *artifacts* to *spatial entities*, such as the `locatedAt` relation in Figure 6.1.

Definition 6.1.21 (Localization relation)**.** A *localization relation* is a *relation* that can be established from an *agent* or an *artifact* to a *spatial entity* to denote that the location of the subject of the relation can be determined with respect to the *spatial characteristics* of the object of the relation.

It is worth to note that the spatial dimension of an STN can be reflected in the digital space by its digital dimension via digital counterparts of spatial entities.

6.1.5 The digital dimension

The digital dimension is represented explicitly in an STN by means of *digital artifacts* hosted by *platforms.* All the information in an STN has to be reified in the digital space via digital artifacts if machines are to access and use it.

For illustrative purposes, Figure 6.2 depicts a digital dimension for the STN representing the David's household (cf. Figure 6.1). We refer to this illustration throughout the rest of this section.

6.1.5.1 Reifying STNs in the digital space

Digital artifacts are *artifacts* that exist in the digital space. As depicted in Figure 6.2, digital artifacts can be the digital counterparts of abstract entities (e.g., `The Doe Group` that represents `The Doe Family`), physical entities (e.g., `Does' Digital House` that represents `Does' House`), or they can exist independently in the digital space, such as a *digital message* (see Definition 6.1.28) sent by `David` to `Sophia`. It is worth to note that an entity can be represented by multiple digital artifacts in various STNs. For instance, David and Sophia can hold *user accounts* (see Definition 6.1.26) on multiple platforms.

Definition 6.1.22 (Digital Artifact)**.** A *digital artifact* is an *artifact* that exists in the digital space.

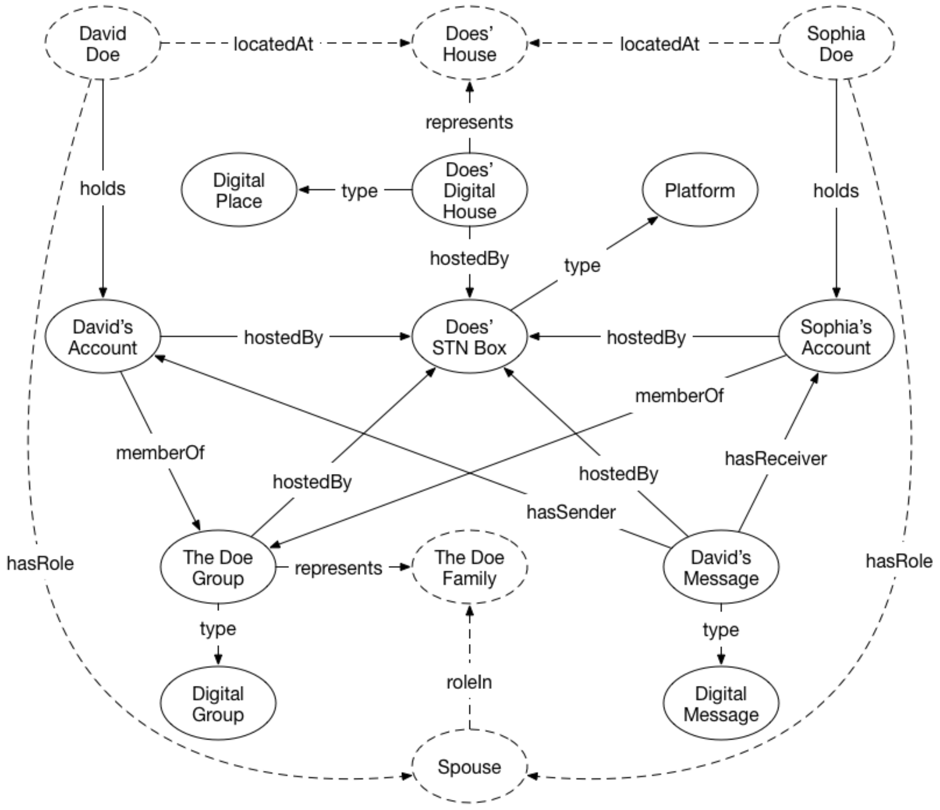


Figure 6.2: An illustration of the digital dimension for Does' home STN. David and Sophia participate in the digital dimension via *user accounts*. Other *digital artifacts* represent digital counterparts of *entities* from other dimensions (cf. Figure 6.1). All the *digital artifacts* in this STN are *hosted* on the *Does' STN Box*.

Platforms provide features to create and manage digital artifacts. Platforms can run, for instance, in the cloud [Armbrust 2010], or on devices close to the edge of the network. The latter is a paradigm referred to as *fog computing* [Bonomi 2012, Vaquero 2014] and provides several benefits in the context of the IoT, such as reduced latency or enabling users to have greater control over their data. For instance, all the digital artifacts depicted in Figure 6.2 are hosted by the `Does' STN Box`, which is a WoT home hub.

Definition 6.1.23 (Platform)**.** A *platform* is an *entity* that represents a collection of features that *agents* can use to create and manage *digital artifacts.*

It is worth to note that, if platforms are to be heterogeneous, clients (e.g., browsers, software agents, multi-agent middleware) must be able to dynamically obtain information about the *operations* (see Definition 6.1.11) supported by a given platform and how they can be implemented via the platform's API. We discuss this further in Chapter 7.

Hosting relations are used to relate *digital artifacts* to their hosting *platforms*, such as the `hostedBy` relations in Figure 6.2. A digital artifact is hosted by a single platform, however, the digital artifacts in an STN can be distributed across multiple platforms. *Hosting relations* play an important role in achieving uniform interfaces in the SWoT by making platforms discoverable. We discuss this further in Chapter 7.

Definition 6.1.24 (Hosting relation)**.** A *hosting relation* is a *relation* that can be established between a *digital artifact* and a *platform* to denote that digital artifact is *hosted by* the platform.

Having introduced the basic entities required to construct the digital dimension of an *STN*, we define an *STN* that has a digital dimension as a *digital STN*.

Definition 6.1.25 (Digital socio-technical network)**.** A *digital socio-technical network* is a *socio-technical network* that is reified in the digital space by means of *digital artifacts* that are *hosted* by *platforms*.

It is worth to emphasize that a *digital STN* can be distributed across multiple *heterogeneous platforms*. Agents can access and use heterogeneous platforms via *artifact-oriented interfaces* (see Section 5.3.1) composed of domain-specific sets of *digital artifacts*.

6.1.5.2 Social artifacts

In Section 5.3.2, we introduced several types of social artifacts, namely *user accounts*, *digital groups*, *digital messages*, *digital places* and *SWoT profiles*. We define the first four in this chapter, and discuss the latter in Chapter 7.

User accounts are digital counterparts of agents. An agent typically performs operations within the context of a platform via a user account, and it is assumed that the entity acting via the user account has been delegated by and acting for the account's holder. David and Sophia, for instance, hold user accounts on their STN Box, which they can use to perform various operations, such as sending messages (cf. Figure 6.2).

Definition 6.1.26 (User Account)**.** A *user account* is a *digital artifact* that represents the digital counterpart of an *agent*.

It is worth to note that *social relations* in a *digital STN* can be established between *agents* or *user accounts* they hold. For instance, in FOAF [Brickley 2014] relations are established between *agents*, and on Twitter[1] relations are created between *user accounts*. An agent can hold multiple user accounts, for instance, in various domain-specific STNs and publish domain-specific data to each of them, in which case it is useful for another agent to keep track only of the individual user accounts that are of interest.

Remark 6.1.1. *Social relations* in a *digital STN* can be established between *agents* or *digital artifacts* that represent them.

[1]http://www.twitter.com/

Digital groups are digital counterparts of *groups*. Digital groups, however, may provide a limited model of the represented group. For instance, in Figure 6.2, `The Doe Group` does not represent the various roles enacted by members of `The Doe Family`. It is also worth to note that `David Doe` and `Sophia Doe` are *members of* `The Doe Group` via their *user accounts*. Therefore, *The Doe Group* may not be an accurate representation of *The Doe Family* if, for instance, David and Sophia have children that are not registered on the `Does' STN Box`. Furthermore, digital groups can be digital counterparts of groups that are not represented explicitly in the STG. This could be the case, for instance, for a *group* that is created as an ad-hoc *digital group* in an online social network.

Definition 6.1.27 (Digital Group)**.** A *digital group* is a *digital artifact* that represents the digital counterpart of a *group*.

Digital messages are the digital counterparts of *messages* (see Definition 6.1.16) reified and disseminated in the digital space. For instance, a spoken message in the physical world can be reified and disseminated in various forms in the digital space, such as via an audio recording or reproduced as text. *Digital messages* can be attached to *artifacts*, such as a reply to a previous *digital message* or a "comment" on a *digital artifact*. Digital messages are typically transmitted via a user account, and the recipients can be agents, user accounts or groups. For instance, the STG in Figure 6.2 represents a *message* transmitted by David to Sophia via their *user accounts*.

Definition 6.1.28 (Digital Message)**.** A digital message is a digital artifact that represents a piece of information an agent can disseminate to a set of agents, user accounts and/or digital groups.

It is worth to note that a digital message can be *transmitted* or *shared* in the digital space. In the former case, a copy of the original message is sent to the recipients (e.g., a direct message sent via Facebook), while in the latter the digital message is only made available to the recipients, possibly for a limited period of time (e.g., a direct message sent via Twitter). That is to say, the creator of a shared digital message retains the control over its lifecycle, and for instance may delete the digital message or modify its sharing settings at any time.

Digital places are digital counterparts of *places* in the physical world. For instance, in Figure 6.2 the `Does' Digital House` represents `Does' House`. Similar to digital groups, the actual place may or may not be represented explicitly in the STG.

Definition 6.1.29 (Digital Place)**.** A *digital place* is a *digital artifact* that represents a *place* in the physical world.

We provide formal definitions for the social artifacts introduced in this section in Section 6.3.

6.2 Formal Definitions

In the previous section, we introduced entities that can be part of an STN and we organized them in multiple dimensions. We now formalize our discussion. Per the generality principle (see Section 5.2.2.1), in this section we provide a mathematical model that can be used to describe STNs at a high level of abstraction. To this purpose, we formalize the primary definitions introduced in Section 6.1.1 and the *social relations* introduced in Section 6.1.2. The mathematical model introduced in this section can be further specialized to model more specific types of STNs. In Section 6.3, for instance, we extend our definitions to model *digital STNs* (see Definition 6.1.25).

In what follows, we structure our discussion in a top-down manner: we first introduce a formal definition for STNs, and then we further discuss and define each element of this definition. Throughout the rest of this section, we use and expand on our previous example illustrated in Figure 6.1.

6.2.1 Structure

We use $\mathcal{E}$ to denote the set of *entities* (see Definition 6.1.1) in the SWoT. We use $\mathcal{AG} \subset \mathcal{E}$ to denote the set of *agents* (see Definition 6.1.6), $\mathcal{ET} \subset \mathcal{E}$ to denote the set of *entity types* (see Definition 6.1.4), and $\mathcal{RT} \subset \mathcal{E}$ to denote the set of *relation types* (see Definition 6.1.3), where $\mathcal{AG}$, $\mathcal{ET}$, and $\mathcal{RT}$ are mutually disjoint.

We have informally defined an *STN* as a dynamic and regulated system of *agents* and other *entities* interrelated in a meaningful manner via *relations* (cf. Definition 6.1.9).

Definition 6.2.1 (Socio-technical network)**.** Let $\mathcal{S}$ denote the set of *STNs*. An STN $s \in \mathcal{S}$ is a structure that includes:

- G_t: the *STG* of s, that is the state of s at any given point in time t (see Definition 6.1.10);
- *Ops*: the set of *operations* (see Definition 6.1.11) supported by s;
- *Norms*: the set of *norms* (see Definition 6.1.17) that regulate the use of *operations* in *Ops*;
- O: an ontology expressed in a given knowledge representation language L that encapsulates domain-specific knowledge for s.

Per the generality principle (see Section 5.2.2.1), we do not impose unnecessary restrictions on the knowledge representation language used by the ontology of an STN. For instance, an STN can use a description logic ontology or a first-order predicate logic ontology. However, we apply constraints on the knowledge that is encapsulated in the ontology. For instance, we say that the ontology of an STN defines the *entity types* (see Definition 6.1.4) and *relation types* (see Definition 6.1.3) used within the STN.

Definition 6.2.2 (Entity and relation types)**.** The signature of the ontology of an *STN* $s \in \mathcal{S}$ defines two sets of non-logical symbols: $ETypes \subseteq \mathcal{ET}$, which denotes the *entity types* that can be assigned to *entities* in s, and $RTypes \subseteq \mathcal{RT}$, which denotes the *relation types* that can be assigned to *relations* among *entities* in s.

The *minimal* vocabulary of an STN defines an *entity type* for *agents*, a *relation type* that denotes *typing relations* (see Definition 6.1.5), and a *relation type* that denotes *social relations* (see Definition 6.1.13). Throughout the rest of this chapter, symbols for *entity types* are written in "camel case" and always begin with a capital letter.

Definition 6.2.3. For all $s \in \mathcal{S}$, we define symbols $Agent \in ETypes$ and $\{type, connectedTo\} \subset RTypes$, where *'type'* denotes *typing relations* that associate *entity types* from *ETypes* to *entities* in s, and *'connectedTo'* denotes *social relations* established between two *agents.*

We can now formally define the STG of an STN as follows:

Definition 6.2.4 (Socio-technical graph)**.** The STG G_t of a given STN $s \in \mathcal{S}$ is a directed edge-labeled multigraph given by:

$$G_t = (N, E, s_E, t_E, l_E), \tag{6.1}$$

where:

- $N \subset \mathcal{E}$ is a set of nodes that represent interrelated *entities* in s at time t;
- $E \subset \mathcal{E}$ is a set of directed edges that represent *relations* in s at time t;
- $s_E : E \to N$ is a function that maps an edge in E to a node in N that denotes the source of a *relation* in s at time t;
- $t_E : E \to N$ is a function that maps an edge in E to a node in N that denotes the target of a *relation* in s at time t;
- $l_E : E \to RTypes$ is a labeling function that maps an edge in E to a *relation type* in *RTypes* that denotes the type of a *relation* in s at time t.

Notation 6.2.1. For clarity, henceforth we denote the directed labeled edges in the STG of an STN $s \in \mathcal{S}$ at time t as triples of the form $(subject, relation\_type, object) \in N \times RTypes \times N$.

For example, in our illustration in Figure 6.1, David and Sophia are agents and spouses. We define an STN $s_1 \in \mathcal{S}$ and choose an ontology O_{s_1} such that it defines the entity type *Agent* and three relation types, namely *type*, *connectedTo*, and *spouseOf*. Given two individuals, *DavidDoe* and *SophiaDoe*, Listing 6.1 shows the STG obtained using these definitions.

$$\begin{aligned}
&ETypes_{s_1} = \{Agent\}\\
&RTypes_{s_1} = \{type, connectedTo, spouseOf\}\\
&G_{t,s_1} = (\{SophiaDoe, DavidDoe, Agent\},\\
&\qquad \{(SophiaDoe, type, Agent), (DavidDoe, type, Agent),\\
&\qquad (DavidDoe, spouseOf, SophiaDoe), (SophiaDoe, spouseOf, DavidDoe)\})
\end{aligned}$$

Listing 6.1: A partial formalization of Does' home STG in Figure 6.1.

The above example also motivates the need for *autonomous reasoning* in STNs. Agents, for instance, should be able to autonomously infer from knowledge encapsulated by O_{s_1} that if David and Sophia are *spouses*, then there exist *social relations* between them. That is to say, a *spouseOf* relation implies a *connectedTo* relation. Agents would then be able to reliably interpret the STG in Listing 6.1 without hard-coding processing logic specific to *spouseOf* relations.

6.2.2 Inferences

Agents can infer knowledge in an STN based on the STN's *ontology* and *STG* (see Definition 6.2.1). We express this formally as follows:

Definition 6.2.5. Given $s \in \mathcal{S}$, we write $O \cup G_t \vDash_L (a, rel, b)$, where $a, b \in N$ and $rel \in RTypes$, to say that we use knowledge encapsulated in O and G_t to infer that (a, rel, b) holds in s.

We expand on our previous example and say that we choose O_{s_1} to be a description logic ontology, such as an OWL ontology, that defines *spouseOf* to be symmetric and $spouseOf \sqsubseteq connectedTo$. From O_{s_1} and the STG in Listing 6.1 it can thus be inferred that:

$$O_{s_1} \cup G_{t,s_1} \vDash_{DL} (SophiaDoe, connectedTo, DaveDoe)$$
$$O_{s_1} \cup G_{t,s_1} \vDash_{DL} (DaveDoe, connectedTo, SophiaDoe)$$

Listing 6.2: Knowledge inferred from O_{s_1} and the STG in Listing 6.1.

We introduce two notations that we use throughout the rest of this chapter, namely for entities of a given *entity type* and relations of a given *relation type*.

Notation 6.2.2. We denote the restriction of N to all nodes of a given type $T \in ETypes$ by $N[T] = \{n \in N \mid O \cup G_t \vDash_L (n, type, T)\}$.

Notation 6.2.3. We denote the restriction of all edges of a given type $rel \in RTypes$ by $R_{rel} = \{(a, b) \in N \times (N \cup ETypes) \mid O \cup G_t \vDash_L (a, rel, b)\}$.

For instance, from Listing 6.1 and Listing 6.2, we use these notations to write:

$$N_{s_1}[Agent] = \{SophiaDoe, DavidDoe\}$$
$$R_{connectedTo,s_1} = \{(SophiaDoe, DavidDoe), (DavidDoe, SophiaDoe)\}$$
$$R_{type,s_1} = \{(SophiaDoe, Person), (SophiaDoe, Agent), (DavidDoe, Person), (DavidDoe, Agent)\}$$

6.2.3 Dynamics

Agents can modify the state of an STN, that is to say its STG, by means of *operations* (see Definition 6.1.11). An *operation* is performed by a single *agent* within the scope of an *STN*. The operation is applied to the *STG* of the STN and it can have a set of input parameters. The operation may modify the STG, in which case it defines a transition from one state of the STN to another. In addition, the operation can also have a set of output parameters, for instance, to return the result of the operation.

Notation 6.2.4. We use $\mathcal{G}$ to denote the set of all possible *STGs*.

Definition 6.2.6 (Operation)**.** An *operation* $op \in Ops$ within the scope of a given STN $s \in \mathcal{S}$ is a function of the form:

$$op : \mathcal{G} \times \mathcal{AG} \times 2^{Input} \rightarrow \mathcal{G} \times 2^{Output}, \tag{6.2}$$

where *Input* and *Output* are technical notations for the sets of all possible input and output parameters of an operation.

For example, David acquires a social wristband, which we interpret as an atomic activity performed by David. The operation of *acquiring a social thing* can be applied to the STG G_{t_1,s_1} in Listing 6.1 with one input parameter, such as the entity that represents the acquired social wristband, and result in an STG G_{t_2,s_1} in which the social thing is owned by David.

6.2.4 Norms

Norms (see Definition 6.1.17) in an STN can be used to *regulate the autonomous behavior of agents* via deontic statements (i.e., obligations, permissions, prohibitions). Research in multi-agent systems proposes several models that can be used to define and formalize norms (see Section 4.3). Extracting the commonality from the various existing models to the aim of providing general, formal definitions for normative concepts is a large undertaking that we leave as future work. This, however, does not influence our contributions or the clarity of the rest of this dissertation.

In addition to regulating the behavior of agents, and in accordance with our formal model, we consider that *norms* can also be used to *regiment the output of operations* in *digital STNs* (see Definition 6.1.25). We discuss digital STNs in Section 6.3. Per Definition 6.2.6, regimenting the output of an operation in a *digital STN* implies that norms can affect the set of output parameters returned by a *platform* (see Definition 6.1.23). For instance, the Does can apply privacy policies to their STN Box (see Section 6.1.5) such that it advertises the things they own only to people to which the Does have *social relations*, that is *connectedTo* relations (cf. Definition 6.2.2). In other words, two different people, for instance, can obtain two different profiles of the Doe family within the Does' home STN based on the STG of the STN and applicable privacy policies.

6.3 Digital Socio-technical Networks

The general mathematical model for STNs introduced in the previous section can be further extended and specialized to satisfy domain-specific requirements, for instance, by defining new *entity types*, *relations types*, and *operation types*. In this section, we extend and specialize our general STN model to define digital STNs (see Definition 6.1.25) of persons and *social things* (see Definition 6.1.7). The *digital STN model* we introduce in this section formalizes all the entities discussed in Section 6.1.

6.3.1 Entity and relation types

A digital STN is reified in the digital space by means of *digital artifacts* (see Definition 6.1.22) that are *hosted* (see Definition 6.1.24) by *platforms* (see Definition 6.1.23). *Digital artifacts* may *represent entities*, such as persons or *social things* (see Definition 6.1.7). Persons can *own social things*. We formalize this discussion in what follows.

Definition 6.3.1 (Digital socio-technical network)**.** Let $\mathcal{DS} \subset \mathcal{S}$ be the set of *digital STNs*. Given an STN $s \in \mathcal{S}$, then $s \in \mathcal{DS}$ if and only if $\{\mathit{Entity}, \mathit{Person}, \mathit{SocialThing}, \mathit{DigitalArtifact}, \mathit{Platform}\} \subset \mathit{ETypes}$, $\{\mathit{represents}, \mathit{hostedBy}, \mathit{owns}\} \subset \mathit{RTypes}$, and it can be entailed from the ontology of s that:

- $N[\mathit{Agent}], N[\mathit{DigitalArtifact}]$, $N[\mathit{Platform}]$ are mutually disjoint subsets of $N[\mathit{Entity}]$;
- $N[\mathit{Person}] \cup N[\mathit{SocialThing}] \subset N[\mathit{Agent}]$, and $N[\mathit{Person}] \cap N[\mathit{SocialThing}] = \varnothing$.

Notation 6.3.1. We use $\mathcal{DA} \subset \mathcal{E}$ to denote the set of *digital artifacts*, and $\mathcal{P} \subset \mathcal{E}$ to denote the set of *platforms*.

Definition 6.3.2 (Representation relation)**.** Given $s \in \mathcal{DS}$, $d \in \mathcal{DA}$ and $e \in \mathcal{E}$, $R_{\mathit{represents}}$ is a partial function $R_{\mathit{represents}} : N[\mathit{DigitalArtifact}] \nrightarrow N[\mathit{Entity}]$ and we say that 'd represents e' if and only if $(d, e) \in R_{\mathit{represents}}$.

The binary relation $R_{\mathit{represents}}$ is defined as a partial function, which implies that digital artifacts may exist independently in the digital space, without being the digital counterpart of a physical or abstract entity (cf. Definition 6.1.22). We formalize the $\mathit{hostedBy} \in \mathit{RTypes}$ relation in a similar manner.

Definition 6.3.3 (Hosting relation)**.** Given $s \in \mathcal{DS}$, $d \in \mathcal{DA}$ and $p \in \mathcal{P}$, R_{hostedBy} is a function $R_{\mathit{hostedBy}} : N[\mathit{DigitalArtifact}] \rightarrow N[\mathit{Platform}]$ and we say that 'd is hosted by p' if and only if $(d, p) \in R_{\mathit{hostedBy}}$.

The fourth type of relation introduced in Definition 6.3.1 (Digital STN), $\mathit{owns} \in \mathit{RTypes}$, represents a directed ownership relation from an *entity*, such as an *agent* or a *group* of agents, to a *social thing*.

Definition 6.3.4 (Ownership relation)**.** Given $s \in \mathcal{DS}$, R_{owns} is a binary relation $R_{\mathit{owns}} \subseteq N \times N[\mathit{SocialThing}]$ such that $O \vDash_L (n, m) \in R_{\mathit{owns}} \Rightarrow (n, m) \in R_{\mathit{connectedTo}}$, and we say that an entity $e \in N$ 'owns' a social thing $t \in N[\mathit{SocialThing}]$ if and only if $(e, t) \in R_{\mathit{owns}}$.

6.3.2 Operation types

In a *digital STN*, *operations* are typically performed via interfaces provided by *platforms*, and they typically depend on the various *digital artifacts* available within the STN. Per the generality principle (see Section 5.2.2.1), our aim is to support platform heterogeneity and thus avoid the over-specification of operations via unnecessary restrictions.

The general form of an *operation* is given by Definition 6.2.6 in Section 6.2.3: all operations are performed by an *agent*, they are applied to an *STG*, they may have *input* and *output parameters*, and they may have side effects on the STG. We specialize this general form of an *operation* to define *operation types* (see Definition 6.1.12) via adding constraints, such as required *input* parameters, expected *output* parameters, or side effects an *operation* of a given *operation type* must have on the *STG* on which it is applied. The various constituents we use to define *operation types* are presented in Table 6.1. Henceforth, we also use a special notation to refer to an *agent* performing an *operation*:

Notation 6.3.2. Henceforth, we use a_p to denote an *agent* performing an *operation.*

Consitutent	**Description**
Desc	A natural language description of a class of operations.
Input	Required input parameters. Implementers may define additional required or optional input parameters.
Preconds	A set of conditions that must hold for applying this type of operation.
Postconds	A set of conditions that must hold if the operation is completed successfully.
Output	Suggested output of a successful operation.

Table 6.1: Constituents used to define classes of operations.

The suggested output of the *operation types* we define for *digital STNs* is generally a subgraph of the *STG* on which the *operation* is performed that represents a description of a digital artifact within the STN. The returned description may be specific to the *platform* that is hosting the *digital artifact* and may be affected by norms, such as privacy policies (see Section 6.2.4).

Definition 6.3.5. The description of a *digital artifact* $d \in N[\mathit{DigitalArtifact}]$ in a *digital STN* $s \in \mathcal{DS}$, denoted desc_s, returns a set of edges in which d is either the source or the target:

$$\mathit{desc}_s : N[\mathit{DigitalArtifact}] \to 2^E,\ \mathit{desc}_s(d) = \{(a, \mathit{rel}, b) \in E \mid a = d \lor b = d\}.$$

In what follows, we define two fundamental *operation types* for *creating* and *deleting social relations* (see Definition 6.1.13), that is *relations* to other *agents. Social relations* can exist between *agents* or their digital counterparts (see Remark 6.1.1).

We define an *operation type* for creating an outgoing relation from a performing agent a_p to a given target as follows:

CreateRelationTo :

```
1. Desc: Agent $a_p$ performs this operation to create a relation from
   itself (or a digital artifact that represents it) to a targeted
   entity $u$, where $u$ is an agent or a digital artifact that represents
   an agent.
2. Input: $p \in \mathcal{P}, u \in \mathcal{AG}^*_{\mathcal{S}}$.
3. Preconds: $u \in N \lor \exists\, u' \in N$ s.t. $u' \in \{a_p\}^*_s$.
4. Postconds: $u \in N \land \exists\, u' \in \{a_p\}^*_s$ s.t. $u' \in N \land (u', u) \in R_{connectedTo}$.
5. Output: $desc_s(u)$.
```

The performing *agent* a_p is the implicit source of the intended *relation*, while the target, a required input parameter, is either an *agent* or a *digital artifact* that represents an *agent*. The precondition requires that either the source or the target of the intended *relation* must already be part of the *STN* in which the *operation* is performed. This precondition implies that *agents* within a given STN s can add *relations* to targets on other *STNs*, or that *agents* outside of s can add relations to targets within s, thus enabling *linking across STNs*. The postcondition specifies that successfully completing this type of *operation* implies the resulting *STG* contains a *connectedTo* edge between the source and the target, and both the source and the target have been added to the set of nodes (if the case). The suggested output of a *CreateRelationTo* operation is a description of the target. Implementers may choose to further specialize this type of operation within their *STNs*, for instance by requesting that both the source and the target of the relation are already part of the *STN*.

We define an *operation type* for deleting an outgoing relation from a performing *agent* a_p to a given target in a similar fashion:

```
DeleteRelationTo:
1. Desc: Agent $a_p$ performs this operation to delete a relation from
   itself (or a digital artifact that represents it) to a targeted
   entity $u$, where $u$ is an agent or a digital artifact that represents
   an agent.
2. Input: $p \in \mathcal{P}, u \in \mathcal{AG}^*_{\mathcal{S}}$.
3. Preconds: $\exists\, u' \in \{a_p\}^*_s$ s.t. $(u', u) \in R_{connectedTo}$.
4. Postconds: $\nexists u' \in \{a_p\}^*_s$ s.t. $(u', u) \in R_{connectedTo}$.
5. Output: $desc_s(u)$.
```

In the above specification, the precondition for deleting a *social relation* is that the relation exists, and the postcondition is that the *social relation* has been deleted. The suggested output of this type of operations returns a description of the target.

Agents can crawl *STNs* via *social relations*. We define an *operation type* for retrieving the outgoing *social relations* of a targeted *agent* or *user account* as follows:

```
GetOutgoingRelations:
1. Desc: Agent $a_p$ performs this operation to retrieve the outgoing
   social relations of a targeted entity $u$, where $u$ is an agent or a
   user account.
2. Input: $p \in \mathcal{P}, u \in \mathcal{AG}^*_{\mathcal{S}}$.
3. Preconds: $(u, p) \in R_{hostedBy} \lor \exists\, u' \in \{u\}^*_s$ s.t. $(u', p) \in R_{hostedBy}$.
4. Postconds: None.
5. Output: $\{c \in N[UserAccount] \mid (u, c) \in R_{connectedTo} \lor (u', c) \in R_{connectedTo}\}$.
```

The precondition for retrieving the outgoing *social relations* of a targeted *agent* is that there is a *digital artifact* that represents the *agent* and is hosted by platform p serving the request. This *operation type* does not impact the *STN*, therefore there are no preconditions. If the operation is successful, it should return a graph containing the descriptions of the *entities* to which the targeted *agent* has outgoing *social relations.* In Chapter 8, we present how this *operation type* is implemented by various existing social platforms, such as Facebook and Twitter.

Operation types for creating, retrieving, and deleting incoming social relations can be defined similarly.

6.3.3 Social artifacts

In this section, we provide formal definitions for the *social artifacts* introduced in our discussion of the digital dimension of an STN (see Section 6.1.5.2). For each social artifact, we define artifact-specific *entity types*, *relation types*, and *operation types.* It is worth to note that our model can be easily extended further with other types of artifacts in a similar manner.

6.3.3.1 User accounts

User accounts are *held* by *agents* and *represent* their digital counterparts in the SWoT (see Definition 6.1.26). We define the *entity types*, *relation types*, and *operation types* for representing and using *user accounts* in what follows.

Definition 6.3.6 (Entity and relation types for user accounts)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *user accounts*, the ontology of s defines $UserAccount \in ETypes$ and $heldBy \in RTypes$ such that it can be entailed that:

- any node that is a *user account* is also a *digital artifact*:
  $\forall\, n \in \mathcal{E}, n \in N[UserAccount] \Rightarrow n \in N[DigitalArtifact]$;
- a *user account* is *held by* a single *agent* and it *represents* the *agent*:
  $R_{heldBy} : N[UserAccount] \rightarrow N[Agent]$,
  $\forall\, s \in \mathcal{S}, O \vDash_L (u, a) \in R_{heldBy} \Rightarrow (u, a) \in R_{represents}$.

Definition 6.3.7 (Operation types for user accounts)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *user accounts*, the set of operations of s may include the following *operation types*: $\{GetUserAccount, WhoIsAgent, CreateUserAccount, UpdateUserAccount, PatchUserAccount, DeleteUserAccount\} \subset Ops$.

An *operation* of the *GetUserAccount operation type* is used to retrieve a description of a *user account* from an *STN* hosted on a given *platform.* This *operation type* requires two input parameters: the *platform* on which the *operation* is to be performed and the *user account* to be retrieved. The *agent* a_p performing the *operation* is implicit (see Notation 6.3.2). The *precondition* for an *operation* of this type to be completed successfully is that the *user account* is indeed *hosted by* the *platform.* This *operation type* has no side effects on the state of the *STN.* If an *operation* of this type is successful, the suggested output for this *operation type* is the *description* (see Definition 6.3.5) of the requested *user account*.

```
GetUserAccount:
1. Desc: Agent $a_p$ retrieves the description of a user account on
    platform $p$.
2. Input: $p \in \mathcal{P}, u \in N[UserAccount]$.
3. Preconds: $(u, p) \in R_{hostedBy}$.
4. Postconds: None.
5. Output: $desc_s(u)$.
```

An *operation* of the *WhoIsAgent operation type* is used to retrieve the *user accounts* that are *held by* an *agent* within an STN an *hosted by* a given *platform*. This *operation type* requires two input parameters, namely the *platform* and the targeted *agent*, and has no preconditions or postconditions. The suggested output for this *operation type* is a set of *user accounts* held by the targeted *agent* and *hosted by* the given *platform*.

```
WhoIsAgent:
1. Desc: Agent $a_p$ retrieves the user accounts of an agent $a$ that are
    hosted on platform $p$.
2. Input: $p \in \mathcal{P}, a \in N[Agent]$.
3. Preconds: None.
4. Postconds: None.
5. Output: $\{u \in N[UserAccount] \mid (u, a) \in R_{heldBy}\}$.
```

An *operation* of the *CreateUserAccount operation type* is used to create a *user account* on a given *platform*. This *operation type* requires one input parameter, the *platform* on which the *user account* is to be created, and it has the precondition that a unique *digital artifact*, that is to say a *digital artifact* not already in use, can be "chosen" (i.e., created). If an *operation* of this *operation type* is completed successfully, a new *UserAccount* artifact is created, and it is *held by* the agent performing the operation (i.e., a_p) and hosted by *platform* on which the *operation* is performed. The suggested output is the *description* of the created *user account*. It is worth to emphasize that it is possible for implementers to further specialize this *operation type*, for instance, to require additional input parameters, such as a name to be displayed within the *STN* for the created *user account*.

```
CreateUserAccount:
1. Desc: Agent $a_p$ creates a user account on platform $p$.
2. Input: $p \in \mathcal{P}$.
3. Preconds: $\exists\, d \in \mathcal{DA}$ s.t. $d \notin N[DigitalArtifact]$.
4. Postconds: $d \in N[UserAccount] \wedge (d, a_p) \in R_{heldBy} \wedge (d, p) \in R_{hostedBy}$.
5. Output: $desc_s(d)$.
```

We define two *operation types* for updating *descriptions* of *user accounts*, namely *UpdateUserAccount* and *PatchUserAccount*. The former is used to replace the entire *description* of a *user account* with the one provided as an input parameter. The latter is used to delete and add sets of edges to the *description* of a *user account*. It is worth to note that we have chosen to define these two *operation types* to resemble the semantics of the HTTP `PUT` [Fielding 2014c] and `PATCH` [Dusseault 2010] methods, which ensures a more transparent mapping of these *operation types* to both HTTP and CoAP (see Section 2.4 for details on the Web of Things and enabling technologies). Implementers, however, have the possibility to extend our *digital STN model* with additional *operation types*.

UpdateUserAccount
1. Desc: Agent a_p updates the description of a user account hosted on platform p.
2. Input: $p \in \mathcal{P}, u \in N[UserAccount], D \subset \mathcal{E} \times RTypes \times \mathcal{E}$.
3. Preconds: $(u, a_p) \in R_{heldBy} \wedge (u, p) \in R_{hostedBy}$.
4. Postconds: $desc_s(u) = D$.
5. Output: $desc_s(u)$.

PatchUserAccount
1. Desc: Agent a_p patches the description of a user account hosted on platform p.
2. Input: $p \in \mathcal{P}, u \in N[UserAccount], D_1 \subset E, D_2 \subset \mathcal{E} \times RTypes \times \mathcal{E}$.
3. Preconds: $(u, a_p) \in R_{heldBy} \wedge (u, p) \in R_{hostedBy}$.
4. Postconds: $E = (E \setminus D_1) \cup D_2$.
5. Output: $desc_s(u)$.

An *operation* of the *DeleteUserAccount operation type* is used to remove a *user account* and its associated *description*, which are *hosted by* a given *platform*, from an *STN*.

DeleteUserAccount:
1. Desc: Agent a_p deletes a user account on platform p.
2. Input: $p \in \mathcal{P}, u \in N[UserAccount]$.
3. Preconds: $(u, a_p) \in R_{heldBy} \wedge (u, p) \in R_{hostedBy}$.
4. Postconds: $u \notin N \wedge E = E\ desc_s(u)$.
5. Output: $desc_s(u)$.

6.3.3.2 Digital Groups

Digital groups (see Definition 6.1.27) are digital counterparts of *groups* (see Definition 6.1.14).

Definition 6.3.8 (Entity and relation types for digital groups)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *digital groups*, the ontology of s defines $\{Group, DigitalGroup\} \subseteq ETypes$ and $memberOf \in RTypes$ such that it can be entailed that:

- any node that is a *digital group* is also a *digital artifact*:
  $\forall\, n \in \mathcal{E}, n \in N[DigitalGroup] \Rightarrow n \in N[DigitalArtifact]$;

- *agents* can be *members of groups*, and *user accounts* can be *members of digital groups*:
  $R_{memberOf} \subseteq (N[Agent] \times N[Group]) \cup (N[UserAccount] \times N[DigitalGroup])$.

Definition 6.3.9 (Operation types for digital groups)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *digital groups*, the set of operations of s may include the following *operation types*: $\{\mathit{GetDigitalGroup}, \mathit{CreateDigitalGroup}, \mathit{JoinDigitalGroup}, \mathit{LeaveDigitalGroup}, \mathit{UpdateDigitalGroup}, \mathit{PatchDigitalGroup}, \mathit{DeleteDigitalGroup}\} \subset \mathit{Ops}$.

An *operation* of the *JoinDigitalGroup operation type* is performed by an *agent* to join a *digital group.* This *operation type* requires two input parameters, namely the *platform* on which it is performed and the targeted *digital group*, and has no preconditions. If the operation is successful, a *memberOf* relation whose source is the performing *agent* and whose the target is the *digital group* is added to the STG of the STN.

```
JoinDigitalGroup:
1. Desc: Agent $a_p$ joins a group $g$ on platform $p$.
2. Input: $p \in \mathcal{P}, g \in N[DigitalGroup]$.
3. Preconds: None.
4. Postconds: $(a_p, g) \in R_{memberOf}$.
5. Output: $desc_s(g)$.
```

Other *operation types* can be defined in a similar manner.

6.3.3.3 Digital Messages

Digital messages (see Definition 6.1.28) are digital counterparts of *messages* (see Definition 6.1.16).

Definition 6.3.10 (Entity and relation types for digital messages)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *digital messages*, the ontology of s defines $\{\mathit{Message}, \mathit{DigitalMessage}\} \subseteq \mathit{ETypes}$ and $\{\mathit{hasSender}, \mathit{hasReceiver}, \mathit{replyTo}, \mathit{attachedTo}\} \subseteq \mathit{RTypes}$ such that it can be entailed that:

- any node that is a *digital message* is also a *digital artifact*:
  $\forall\, n \in \mathcal{E}, n \in N[\mathit{DigitalMessage}] \Rightarrow n \in N[\mathit{DigitalArtifact}]$;
- a *digital message* has one sender, which is a *user account*, and one or more receivers, which may be *user accounts* or *digital groups*:
  $R_{\mathit{hasSender}} : N[\mathit{DigitalMessage}] \rightarrow N[\mathit{UserAccount}]$,
  $R_{\mathit{hasReceiver}} \subseteq N[\mathit{DigitalMessage}] \times (N[\mathit{UserAccount}] \cup N[\mathit{DigitalGroup}])$;
- a *message* can be sent in *reply to* other *messages*, and it can be attached to *digital artifacts*:
  $R_{\mathit{replyTo}} \subseteq N[\mathit{DigitalMessage}] \times N[\mathit{DigitalMessage}]$,
  $R_{\mathit{attachedTo}} \subseteq N[\mathit{DigitalMessage}] \times N[\mathit{DigitalArtifact}]$.

Definition 6.3.11 (Operation types for digital messages)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *digital messages*, the set of operations of s may include the following *operation types*: $\{\mathit{GetSentDigitalMessages}, \mathit{GetReceivedDigitalMessages}, \mathit{CreateDigitalMessage}, \mathit{DeleteDigitalMessage}\} \subset \mathit{Ops}$.

An *operation* of the *CreateDigitalMessage operation type* is performed by an *agent* to create a *digital message*. This *operation type* requires as input parameter the *platform* on which the *digital message* is to be created and a set of *recipients*. The precondition is that a unique *digital artifact* can be "chosen" to create the *digital message*. The *digital message* is added to the STG of the STN and appropriate edges are added for the sender and each of the recipients. The suggested output is the *description* of the created *digital message*.

```
$SendMessage$:
1. Desc: Agent $a_p$ sends a message to a set of recipients
   $Recipients \subseteq N[UserAccount] \cup N[DigitalGroup]$.
2. Input: $p \in \mathcal{P}$.
3. Preconds: $\exists\, d \in \mathcal{DA}$ s.t. $d \notin N[DigitalArtifact]$.
4. Postconds: $d \in N[DigitalMessage] \land (d, p) \in R_{hostedBy} \land (d, a_p) \in R_{hasSender} \land$
              $\forall\, r \in Recipients, (d, r) \in R_{hasReceiver}$.
5. Output: $desc_s(d)$.
```

Other *operation types* can be defined in a similar manner.

6.3.3.4 Digital Places

Digital places (see Definition 6.1.29) are digital counterparts of *places* (see Definition 6.1.20).

Definition 6.3.12. Given $s \in \mathcal{DS}$, we define $\{Place, DigitalPlace\} \subseteq ETypes$ and $\{id, name, description, locatedAt\} \subseteq RTypes$, where $R_{locatedAt} \subseteq N[Entity] \times N[Place]$.

Definition 6.3.13 (Entity and relation types for digital places)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *digital places*, the ontology of s defines $\{Place, DigitalPlace\} \subseteq ETypes$ and $locatedAt \in RTypes$ such that it can be entailed that:

- any node that is a *digital place* is also a *digital artifact*: $\forall\, n \in \mathcal{E}, n \in N[DigitalPlace] \Rightarrow n \in N[DigitalArtifact]$;
- *entities* can be *located at places*: $R_{locatedAt} \subseteq N[Entity] \times N[Place]$.

Definition 6.3.14 (Operation types for digital places)**.** In a *digital STN* $s \in \mathcal{DS}$ that supports *digital places*, the set of operations of s may include the following *operation types*: $\{GetDigitalPlace, CheckIntoDigitalPlace, CheckOutOfDigitalPlace, CreateDigitalPlace, UpdateDigitalPlace, PatchDigitalPlace, DeleteDigitalPlace\} \subset Ops$.

An *operation* of the *CheckIntoDigitalPlace operation type* is performed by an *agent* to declare its location at a given *digital place*. This *operation type* requires two input parameters, namely the *platform* on which the operation is executed and the targeted *place*, and it has no preconditions. If an *operation* of this *operation type* is successful, a *locatedAt* relation, whose source is the performing *agent* and target is the *place*, is added to the *STG* of the *STN*. The suggested output is the *description* of the targeted *place*.

```
$CheckIntoDigitalPlace$:
1. Desc: Agent $a_p$ checks into a place.
2. Input: $p \in \mathcal{P}$, $l \in N[DigitalPlace]$.
3. Preconds: None.
4. Postconds: $(a_p, l) \in R_{locatedAt}$.
5. Output: $desc_s(l)$.
```

Other *operation types* can be defined in a similar manner.

6.4 Summary

In this chapter, we addressed Research Question 2, that is *how to model networks of people and autonomous things such that things can manipulate and reason upon them.* To this purpose, we introduced a mathematical model that defines STNs by means of entities interrelated to one another in a meaningful manner. Per the generality principle (see Section 5.2.2.1), our model is *general* such that it can represent heterogeneous STNs and provide an unambiguous, formal foundation for the SWoT. Per the separation of concerns principle (see Section 5.2.2.2), our model is *modular* such that it can be easily extended, for instance, to represent domain-specific STNs. Per the interoperability principle (see Section 5.2.2.3), all domain-specific knowledge in our STN model (e.g., entity types, relation types, operation types) is encapsulated in an easily standardizable form via an ontology.

In Section 6.1, we introduced and illustrated the entities in our model, and we organized them in multiple dimensions that we use to represent STNs. In Section 6.2, we defined a *general* and *extensible* mathematical model for representing *dynamic* and *regulated* networks of *agents* that are interrelated via *social relations.* In Section 6.3, we extended this model to represent *digital STNs*, that is STNs reified in the digital space by means of *digital artifacts* hosted by *platforms.* We provided formal definitions for the *social artifacts* that we use to create *artifact-oriented interfaces* for heterogeneous social and STN platforms (see Section 5.3.2 and Section 6.1.5 for details). Implementers have the possibility to easily extend our general STN model in Section 6.2, and our digital STN model in Section 6.3, in a similar manner to the one presented in this chapter, that is by defining new entity types, relation types, and operation types. Furthermore, in Chapter 7 we provide an OWL ontology that encapsulates all the entities defined by our digital STN model. Implementers can use and extend this ontology to describe heterogeneous *STNs.*

Chapter 7

A Hypermedia-driven Social Web of Things

Contents

In our vision for a *Social Web of Things (SWoT)*, people and things are situated and interact in a *global environment*. The environment is sustained by *heterogeneous platforms* and its main roles include to facilitate *discoverability* and *flexible interaction* in the ecosystem (see Section 5.1.5). In this chapter, we apply the REST architectural style (per Foundational Principle 1) and the *digital STN model* defined in Section 6.3 to create a *hypermedia-driven environment* sustained by *heterogeneous STN platforms*[1]. Hypermedia is central to achieve *uniform interfaces*[2] and thus alleviate the heterogeneity problem in the SWoT, and the social network metaphor

[1]We call an *STN platform* any *platform* (see Definition 6.1.23) that interconnects and enables interaction between people, things, or both. Relations among people and things may or may not be represented explicitly in the form of a *socio-technical graph* (see Definition 6.1.10.

[2]See Section 2.1.1 for further details on the role of hypermedia and the HATEOAS constraint in the REST architectural style.

provides a mechanism to enhance connectivity in the ecosystem and across application domain silos (see Foundational Principle 2). In a hypermedia-driven SWoT, software clients are able to "learn" on-the-fly how to interface with STN platforms in their environment, and can thus transcend platform boundaries to navigate and manipulate the world-wide *socio-technical graph (STG)*. To bring about such an ecosystem, however, we have to address two questions: *how to create hypermedia APIs for STN platforms*, and *how to integrate STN platforms with non-hypermedia APIs into a hypermedia-driven SWoT*.

This chapter is structured as follows. In Section 7.1, we introduce a semantic description framework for STN platforms that conforms to the *digital STN model* defined in Section 6.3. We use this framework in Section 7.2 to address the above questions and provide solutions to achieve *uniform interfaces* for *heterogeneous* STN platforms. In Section 7.3, we structure our approach in a progressive, five-level strategy for the development and integration of STN platforms into the SWoT.

7.1 A Semantic Description Framework for STNs

In this section, we present a framework for creating semantic descriptions of *STNs* (see Definition 6.1.9), and in particular *digital STNs* (see Definition 6.1.25) and their hosting *platforms* (see Definition 6.1.23). We first introduce an OWL ontology [W3C OWL Working Group 2012], which we call the *STN ontology*, that follows our formal definitions in Chapter 6. We then provide descriptions for *agents* (see Definition 6.1.6), *platforms*, and *digital artifacts* (see Definition 6.1.22).

We have chosen OWL 2 [W3C OWL Working Group 2012] because it is a W3C Recommendation (see *interoperability principle* in Section 5.2.2.3) and it is built on top of RDF, which provides a data model that is suitable to represent distributed graphs.

7.1.1 STN ontology

Per the *separation of concerns principle* (see Section 5.2.2.2), the concepts and properties defined by the STN ontology are organized in three modules, as depicted in Figure 7.1:

- *STN-Core*, which provides terms to describe *socio-technical graphs (STGs)* (see Definition 6.1.10); this module is informally aligned with several ontologies, such as the FOAF vocabulary [Brickley 2014], the SIOC Core Ontology [Bojars 2010], or the W3C Geospatial Vocabulary [Lieberman 2007] (cf. Figure 7.1);

- *STN-Operations*, which extends *STN-Core* with terms to describe *operations* (see Section 6.3.2) to be performed on *STGs*;

- *STN-Operations-HTTP*, which extends *STN-Operations* and the W3C HTTP Vocabulary [Koch 2011] to provide terms to describe HTTP-based APIs.

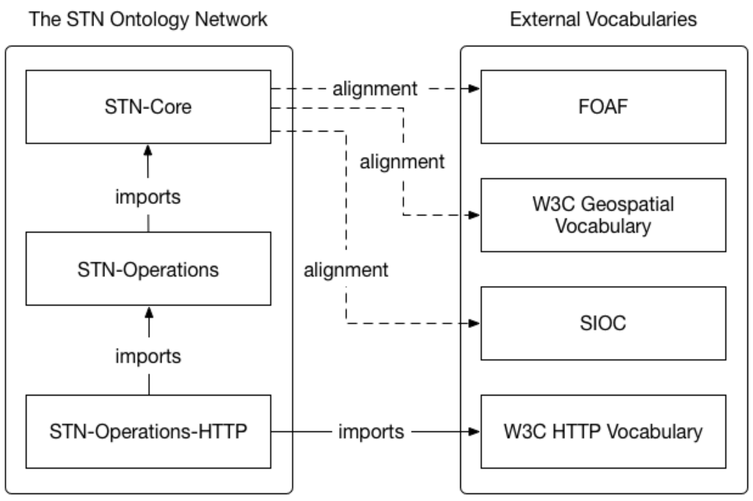


Figure 7.1: The STN ontology network.

This separation of concerns facilitates reusing and extending the STN ontology. For instance, the *STN-Operations-HTTP* module may be substituted with other vocabularies for describing HTTP-based APIs, or even with modules that describe implementations of *operations* by means of other protocols, such as CoAP [Shelby 2014a].

Module	**Prefix**	**Namespace URI**
STN-Core	stn:	http://purl.org/stn/core#
STN-Operations	stn-ops:	http://purl.org/stn/operations#
STN-Operations-HTTP	stn-http:	http://purl.org/stn/operations/http#

Table 7.1: Namespace prefix bindings used throughout the rest of this dissertation for the STN ontology.

We present an overview of each of the three modules in what follows. The complete specifications are available online.[3,4,5] Throughout the rest of this dissertation, for clarity and conciseness, we use the namespace prefix bindings in Table 7.1.

7.1.1.1 STN-Core

STN-Core provides terms that correspond to the *entities* defined in Chapter 6, except the ones related to the *normative dimension* (see Section 6.1.3), which we do not represent in STGs at the moment (see Chapter 10).

Figure 7.2 depicts a simplified overview of *STN-Core* and the various dimensions of an STN that can be described using this module, namely the *social*, *digital*, and

[3]http://purl.org/stn/core/spec
[4]http://purl.org/stn/operations/spec
[5]http://purl.org/stn/operations/http/spec

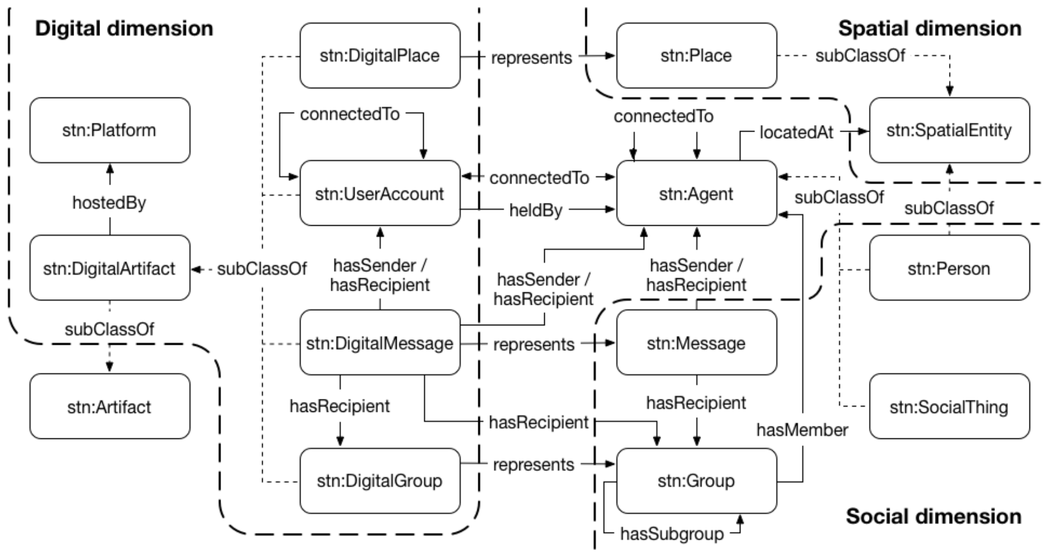


Figure 7.2: A partial overview of STN-Core. For clarity, in this figure we omit the prefixes of properties, that is the `stn:` prefix for the properties defined as part of STN-Core and the `rdfs:` prefix for the `rdfs:subClassOf` property defined as part of RDF Schema.

spatial dimensions (cf. dimensions defined in Section 6.1).

The *social dimension* (cf. Figure 7.2) includes two disjoint classes of *agents* (see Definition 6.1.6), that is *persons* and *social things* (see Definition 6.1.7). *Agents* can be interconnected to one another via *social relations* (see Definition 6.1.13), which are defined as part of *STN-Core* via the `stn:connectedTo` property and can be established either between *agents* themselves, or via their *user accounts*. *Agents* can be members of *groups* (see Definition 6.1.14), and they can exchange *messages* (see Definition 6.1.16) with other *agents* and *groups* of agents.

The *spatial dimension* (cf. Figure 7.2) includes the class of *spatial entities* (see Definition 6.1.19), and its disjoint subclasses of *persons* and *places* (see Definition 6.1.20). *STN-Core* also defines the class of *social things* with spatial coordinates (i.e., that are also *places*), which we refer to as *social places*.[6]

The *digital dimension* includes classes of *digital artifacts* (see Definition 6.1.22) that represent the digital counterparts of *entities* defined by the other dimensions, namely (cf. Figure 7.2): *digital places* (see Definition 6.1.29), *user accounts* (see Definition 6.1.26), *digital messages* (see Definition 6.1.28), and *digital groups* (see Definition 6.1.27). *Digital artifacts* are *hosted by* (see Definition 6.1.24) *platforms* (see Definition 6.1.23).

We illustrate the use of these terms to create *agent descriptions* in Section 7.1.2 and *digital artifact descriptions* in Section 7.1.4.

[6]For clarity, we chose not to represent in Figure 7.2 the class of *social places*, which is denoted by the URI `stn:SocialPlace`.

7.1.1.2 STN-Operations

STN-Operations extends *STN-Core* with terms for describing *operations* (see Definition 6.1.11) supported by *platforms* (see Definition 6.1.23) and *performed by* agents, as depicted in Figure 7.3.

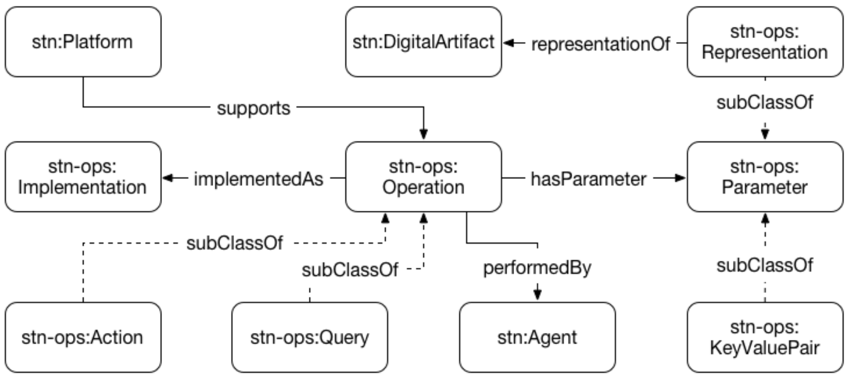


Figure 7.3: A partial overview of STN-Operations. For clarity, we omit the prefixes of the depicted properties.

We introduced the general form of an *operation* in Definition 6.2.6 in Section 6.2. *STN-Operations* defines concepts and properties to describe the *input* and *output parameters* of *operations*. In particular, this module defines two classes of parameters, that is for *representations* and *key-value pairs* (cf. Figure 7.3). The former denotes serialized data objects, such as Turtle [Beckett 2008] representations of *digital artifacts*. The latter is further specialized via several subclasses, such as the class of *user account identifiers* denoted by the `stn-ops:UserAccountID` URI[7], which is used to encapsulate the association between a *platform-specific key* and a *platform-specific identifier* of a user account.

In Section 6.3.2, we defined several *operation types* (see Definition 6.1.12) for *digital STNs*. *STN-Operations* defines concepts for each of these *operation types* and classifies them as either *queries*, which are *operations* used to retrieve information from *STGs*, or *actions*, which are *operations* used to manipulate *STGs* (cf. Figure 7.3). For instance, operations of type *GetUserAccount* (see Section 6.3.3) are *queries* used to retrieve *representations* of *user accounts*, whereas operations of type *CreateUserAccount* (see Section 6.3.3) are *actions* used to create *user accounts*. *Agents* can use this distinction to infer which operations are safe and which operations impact STGs.

Operations can have one or more *implementations* (cf. Figure 7.3). The class of *implementations* defined as part of *STN-Operations* is intended to be extended by other vocabularies concerned with representing operation implementations, such as *STN-Operations-HTTP*.

[7]For clarity, we chose not to represent in Figure 7.3 all the classes of operations or parameters defined as part of STN-Operations. The complete specification is available online: http://purl.org/stn/operations/spec

7.1.1.3 STN-Operations-HTTP

STN-Operations-HTTP extends *STN-Operations* with concepts and properties for translating operations to HTTP requests and extracting RDF data from HTTP responses. That is to say, this module can be used to describe HTTP-based *implementations* of *operations* supported by *platforms*.

This module imports the W3C HTTP vocabulary [Koch 2011] and defines the class of *STN requests* as *HTTP requests* that are also *implementations* of *operations* (cf. Figure 7.4). The class of *STN requests* is also further specialized to denote the class of *authenticated STN requests* (cf. Figure 7.4), which denotes *STN requests* that use *HTTP authentication* [Fielding 2014a].

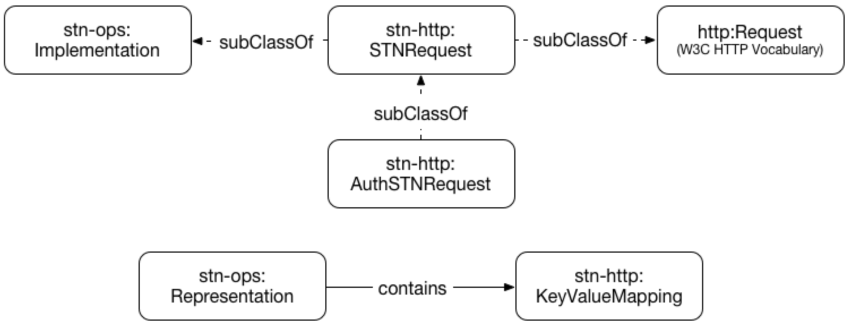


Figure 7.4: A partial overview of STN-Operations-HTTP. For clarity, we omit the prefixes of the depicted properties.

STN-Operations-HTTP defines a class for *key value mappings* that can be *contained* within *representations* (cf. Figure 7.4), and several properties used to associate these mappings both to terms defined by *STN-Core* and platform-specific information. For illustrative purposes, in Section 7.1.4 we show how to use these terms to extract an RDF representation of a *user account* from a JSON representation of a Twitter account.

7.1.2 Agent descriptions

The terms defined by the STN ontology and any of its extensions can be used to create *agent descriptions*, which we call *SWoT profiles* (see Section 5.3.2.2). *SWoT profiles* are thus *RDF graphs* that are part of the *world-wide STG* (see Section 5.3.2). A *SWoT profile* must contain at least one *entity type* (see Definition 6.1.4) for the described *agent* (see Definition 6.1.6), which can be `stn:Agent` or any of its subclasses. In addition, a *SWoT profile* can aggregate general information about the described agent and its *user accounts* (see Definition 6.1.26) hosted by various *platforms* (see Definition 6.1.23).

Definition 7.1.1. (SWoT profile) The *SWoT profile* of an *agent* is an *RDF graph* that describes the *agent* and its *user accounts* in the *SWoT*.

Per Foundational Principle 1 (conformity to REST), agents are uniquely identified in the SWoT, for instance via URIs. In order to support *discoverability*, dereferencing an agent's identifier should retrieve a representation of its *SWoT profile*.

For illustrative purposes, in Listing 7.1 we present a *SWoT profile* for David (see scenarios in Section 5.1). In this example, David Doe is identified by a *hash URI* [Sauermann 2008] that dereferences to David's *user account* (see Definition 6.1.26) hosted by the Does' STN Box, which is an *STN platform*. David's user account also encodes his *SWoT profile*, which could have been generated, for instance, when David registered to the STN Box. It is worth to note, however, that the *SWoT profile* of an agent is not necessarily associated to a user account. A *SWoT profile* can exist, for instance, in a standalone document, similar to a FOAF profile document [Brickley 2014].

```
<http://192.168.0.1/users/david.doe#DavidDoe>
     a stn:Person ;
    stn:name "David Doe" ;
    stn:connectedTo <http://192.168.0.1/users/sophia.doe#SophiaDoe> ;
    stn:holds <http://192.168.0.1/users/david.doe/> ;
    stn:holds [
        a stn:UserAccount ;
        stn:hostedBy
            <http://www.twitter.com/.well-known/stn/#platform> ;
        stn:id "daviddoe" ;
    ] ;
    stn:holds [
        a stn:UserAccount ;
        stn:hostedBy
            <http://www.facebook.com/.well-known/stn/#platform> ;
        stn:id "1550387481863557" ;
    ] .

<http://192.168.0.1/users/david.doe>
    a stn:UserAccount ;
    stn:description "David Doe's user account."@en ;
    stn:hostedBy <http://192.168.0.1/.well-known/stn/#platform> ;
    stn:heldBy <http://192.168.0.1/users/david.doe#DavidDoe> .
```

Listing 7.1: David Doe's *SWoT profile*, which is hosted on the Does' STN Box. David holds *user accounts* on the STN Box, Twitter, and Facebook. David is identified by a *hash URI* [Sauermann 2008] that dereferences to his user account on the STN Box.

Per Listing 7.1, David's *SWoT profile* describes him as an `stn:Person` *named* `David Doe`, who has a *social relation* (see Definition 6.1.13) to another *entity* that is also denoted by a *hash URI*. As the URI might hint to a human, it identifies Sophia Doe, whose *SWoT profile* is also hosted on the Does' STN Box. A software client can obtain this information by dereferencing the URI to retrieve Sophia's *SWoT profile*, which can be defined similarly to the one of David.

David *holds* additional *user accounts* on Twitter and Facebook. Unlike the Does' STN Box, Twitter and Facebook use platform-specific resource identifiers, which are represented via the `stn:id` property (cf. Listing 7.1).[8]

7.1.3 Platform descriptions

Similar to agent descriptions, a *platform description* is an RDF graph created using terms defined by the STN ontology and any of its extensions that encodes information about the features and API of the described *platform* (see Definition 6.1.23). *Platform descriptions* are part of the *world-wide STG* (see Section 5.3.2). Machines can use *platform descriptions* to “learn” on-the-fly how to interface with *platforms*.

Definition 7.1.2. (Platform description) A *platform description* is an *RDF graph* that describes the features and API provided by a *platform*.

Similar to *agent* identifiers, a *platform*’s identifier is uniform and dereferencing it should retrieve a representation of its complete or partial *platform description*. We refer to the document retrieved when dereferencing the URI of an STN platform as its *STN description document*.

For illustrative purposes, in Listing 7.2 we present a *platform description* for the Does’ STN Box (see example in the previous section). From this description, a software client can extract the platform’s name to be displayed to a human user, the base URL of its API, supported authentication protocols (e.g., WebID [Sambra 2015b]), representation formats (e.g., Turtle [Beckett 2008]), and *operations* (see Definition 6.1.11) implemented via its API.

```
@prefix format: <http://www.w3.org/ns/formats/> .

<#platform>
    a stn:Platform ;
    stn:name "Does' STN Box" ;
    stn-http:baseURL "http://192.168.0.1/"^^xsd:anyURI ;
    stn-http:supportsAuth stn-http:WebID ;
    stn-ops:consumes format:Turtle ;
    stn-ops:produces format:Turtle ;
    stn-ops:supports <#createUserAccount> ,
        <#getUserAccount> ,
         (...)
        <#deleteMessage> .
```

Listing 7.2: General platform description of an `stn:STNPlatform` that supports the WebID authentication protocol, Turtle representations, and several *operations* to be performed via its API.

It is worth to note that the interface of the Does’ STN Box is defined in terms of *operations*. We discuss *operation descriptions* in Section 7.1.3.1. However, platforms can also expose interfaces defined in terms of *digital artifacts*. We discuss descriptions of *digital artifact containers* in Section 7.1.3.2. We discuss in further detail the various types of interfaces exposed by STN platforms in Section 7.2.

[8]The URIs provided in this example to identify the Twitter and Facebook platforms are used for illustrative purposes. The URIs should, in fact, point to descriptions of the two platforms (see Section 7.1.3), such as the ones we introduce in Chapter 8.

7.1.3.1 Operation descriptions

An *operation description* is created using terms defined by *STN-Operations* and/or any of its extensions, and must include:

- the *operation type* (see Definition 6.1.12), which is `stn-ops:Operation` or any of its subclasses;
- an *implementation*, which is specified via the `stn-ops:implementedAs` property and is a member of `stn-ops:Implementation` or any of its subclasses;
- descriptions for each *required input parameter*, which are specified via the `stn-ops:hasRequiredInput` property and are members of `stn-ops:Parameter` or any of its subclasses.

In addition, an *operation description* may also include descriptions for *optional input parameters* via the `stn-ops:hasInput` property, and one or multiple *output descriptions* via the `stn-ops:hasOutput` property. The latter are necessary, for instance, to extract representations of digital artifacts from heterogeneous data sources, which we discuss in Section 7.1.4.

For illustrative purposes, in Listing 7.3 we present an *operation description* for a *CreateUserAccount* operation (see formal definition in Section 6.3.3), which is created using terms defined by *STN-Operations* and *STN-Operations-HTTP*. Software clients can perform this operation via an `HTTP POST` to the `/users/` endpoint. The body of the request must include one required input parameter, a name to be displayed within the STN for the created user account, which is a member of the `stn-ops:DisplayedName` parameter class. Optionally, the body of the request may also include a description to be displayed within the STN. The presented *operation description* specifies an output description, which is a Turtle representation of a user account.

```
<#createUserAccount>
    a stn-ops:CreateUserAccount ;
    stn-ops:implementedAs [
            a stn-http:STNRequest ;
            http:methodName "POST" ;
            http:absolutePath "/users/" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:DisplayedName ;
            stn-ops:paramName "displayedName" ;
            stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasInput [
            a stn-ops:Description ;
            stn-ops:paramName "description" ;
            stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasOutput [
        a stn-ops:Representation ;
        stn-ops:mediaType format:Turtle ;
        stn-ops:entityType stn:UserAccount ;
    ] .
```

Listing 7.3: An *operation description* for the `<#createUserAccount>` *operation* supported by the Does' STN Box (see Listing 7.2).

7.1.3.2 Digital artifact container descriptions

Digital artifact containers are *digital artifacts* (see Definition 6.1.22) that may contain other *digital artifacts*. The description of a *digital artifact container* is created using terms defined by *STN-Core* and/or any of its extensions, and must include:

- the *container type*, which is `stn:DigitalArtifactContainer` or any of its subclasses;
- the *platform* hosting the container, which is specified via the `stn:hostedBy` property and is a member of `stn:Platform` or any of its subclasses;
- the *type* of digital artifacts, which is specified via the `stn:memberType` property and is the class denoted by `stn:DigitalArtifact` or any of its subclasses.

Members of a digital artifact container are specified via the `stn:contains` property. It is worth to note that *digital artifact containers* are similar to the *basic containers* defined by the Linked Data Platform (see Section 2.3.3).

For illustrative purposes, in Listing 7.4 we present the description of a *digital artifact container* hosted by the Does' STN Box (see Listing 7.2) and whose members are of type `stn:UserAccount`. This description lists the user accounts of David and Sophia (see Listing 7.1) as members of the described container.

```
<http://192.168.0.1/users/>
    a stn:DigitalArtifactContainer ;
    stn:hostedBy <http://192.168.0.1/.well-known/stn/#platform> ;
    stn:memberType stn:UserAccount ;
    stn:contains <david.doe>, <sophia.doe> .
```

Listing 7.4: Description of a user account container hosted by the Does' STN Box (see Listing 7.2).

7.1.4 Digital artifact descriptions

A *digital artifact description* is created using *STN-Core* and/or any of its extensions. For example, in Listing 7.1 we have already presented a description for David's *user account*.

Definition 7.1.3. (Digital artifact description) The *digital artifact description* of a *digital artifact* is an *RDF graph* that describes the *digital artifact* and its *relations* with other *entities* in the *SWoT*.

Similar to *agent* and *platform* identifiers, dereferencing the uniform identifier of a *digital artifact* should retrieve a representation of its *digital artifact description*.

Some *platforms* (see Definition 6.1.23) may directly produce RDF representations of digital artifacts using the STN ontology. Heterogeneity, however, is central to the SWoT, and other *platforms* may use platform-specific data models and formats. To address this issue, *STN-Operations-HTTP* defines an RDF mapping language that clients can use to extract RDF data from heterogeneous data sources that are modeled via two structures, namely *name/value pairs* and *arrays of values*. These two structures are, for instance, the foundation of the JSON data interchange format [Bray 2014]. Clients could also use a more general RDF mapping language[9], or they could outsource the data integration task to an external service.

For illustrative purposes, in Listing 7.5 we present a mapping created using terms defined by *STN-Operations* and *STN-Operations-HTTP* that extracts an RDF representation of a *user account* from a JSON representation of a Twitter user account produced by the Twitter Public API v1.1[10]. The JSON representation of the Twitter user account contains several *key/value pairs*, such as the Twitter `screen_name` of the user account that is mapped to a platform-specific identifier via the `stn:id` property. Other mappings include the *name* and *description* associated with a Twitter user account, and the *number of incoming/outgoing* social relations from/to other Twitter user accounts. We present more examples for extracting RDF representations from heterogeneous data sources in Chapter 8.

```
<#twitterAccountJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:UserAccount ;
    stn-http:contains [
            a stn-http:KeyValueMapping ;
            stn-http:key "screen_name" ;
            stn-http:STNterm stn:id ;
            stn-http:datatype xsd:string ;
        ] ;
    stn-http:contains [
            a stn-http:KeyValueMapping ;
            stn-http:key "name" ;
            stn-http:STNterm stn:name ;
            stn-http:datatype xsd:string ;
        ] ;
    stn-http:contains [
            a stn-http:KeyValueMapping ;
            stn-http:key "description" ;
            stn-http:STNterm stn:description ;
            stn-http:datatype xsd:string ;
        ] ;
    stn-http:contains [
            a stn-http:KeyValueMapping ;
```

[9] Such as the RDF Mapping Language (RML): http://semweb.mmlab.be/rml/spec.html, Accessed: 27.10.2015

[10] https://dev.twitter.com/overview/api/users, Accessed: 27.10.2015.

```
            stn-http:key "friends_count" ;
            stn-http:STNterm stn:outgoingConnections ;
            stn-http:datatype xsd:integer ;
        ] ;
    stn-http:contains [
            a stn-http:KeyValueMapping ;
            stn-http:key "followers_count" ;
            stn-http:STNterm stn:incomingConnections ;
            stn-http:datatype xsd:integer ;
        ] .
```

Listing 7.5: A mapping for extracting a *digital artifact description* of a *user account* from a JSON representation of a Twitter user account.

7.2 Uniform Interfaces for STN Platforms

We use the STN ontology and the various types of descriptions presented in the previous section to decouple *software clients* from *STN platforms.* Per Foundational Principle 1 (conformity to REST), our aim is to obtain a *uniform interface* that hides platform-specific implementation details and provides standardized access to the SWoT environment. Software clients would then be able to use this interface to *operate across heterogeneous* STN platforms.

The REST architectural style provides guidance on achieving uniform interfaces in distributed hypermedia systems (see Section 2.1.1). Our goal, however, is to be liberal in what platforms may be integrated into the SWoT. For instance, existing Web platforms, such as social and WoT platforms, expose non-hypermedia APIs (see Section 2.1.2). Furthermore, the SWoT may potentially extend to platforms outside of the Web ecosystem. Therefore, a challenge that is important to the successful development and adoption of the SWoT is to integrate *heterogeneous, non-hypermedia interfaces* into a hypermedia-driven environment.

Our approach is based on the *digital STN model* introduced in Section 6.3, which we defined in terms of *digital artifacts* (see Definition 6.1.22) and artifact-specific *operations* (see Definition 6.1.11). To structure our discussion, we classify heterogeneous interfaces of STN platforms based on their mode of interaction with clients guided by the following question: *who performs operations on digital artifacts?* If an interface is defined in terms of *operations* invoked remotely by a client and performed on the STN platform, we say the interface is *control-driven.* If an interface is defined in terms of *digital artifacts* exchanged between a client and the STN platform, we say the interface is *data-driven.* It is worth to note that RESTful interaction is data-driven, whereas most existing Web platforms expose RPC-like APIs that are more similar to control-driven interfaces (see Section 2.1.2).

In what follows, we first discuss REST's uniform interface constraint in the context of the SWoT and propose practical solutions to integration problems. We then apply our discussion to achieve uniform interfaces for *control-driven*, *data-driven*, and *mixed interfaces* of STN platforms. Our discussion in these last three sections focuses on the integration of non-hypermedia STN platforms into the SWoT.

7.2.1 Uniformity constraints

We reformulate REST's interface constraints (see Section 2.1.1) in the context of the SWoT as follows:

C1 digital artifacts are uniformly identified in the SWoT;

C2 digital artifacts are manipulated via representations;

C3 messages exchanged between components are self-descriptive;

C4 interaction between clients and STN platforms is driven by hypermedia.

We discuss each of these interface constraints in further detail in what follows.

7.2.1.1 Identification of digital artifacts

Per C1, *digital artifacts* must be uniformly identified in the SWoT such that they can be referenced globally and independent of context. On the Web, resources are uniformly identified by means of URIs. The APIs of existing Web platforms, however, typically use platform-specific identifiers to identify resources (see Section 2.1.2). Therefore, in order to integrate into the SWoT environment *digital artifacts* identified within the context of an STN platform, it is necessary to encapsulate the context within the reference to the *digital artifact*. We propose that a *digital artifact* is uniformly identified in the SWoT by a URI [Berners-Lee 2005], or it is identified by a tuple given by the URI of its hosting *STN platform* and a platform-specific identifier of the *digital artifact*. Per our discussion in Section 7.1.3, dereferencing the identifier of an *STN platform* should return a *platform description* that clients can use to access the platform and retrieve the referenced *digital artifact*.

For illustrative purposes, in Listing 7.1 (David's SWoT profile), David's user account on his STN Box is identified via an HTTP URI, and his user accounts on Twitter and Facebook can be identified via the platforms' HTTP URIs and platform-specific identifiers. For the former, clients can directly dereference the HTTP URI of David's user account via an `HTTP GET` request to retrieve a usable representation from David's STN Box. For the last two user accounts, however, clients would first have to dereference the HTTP URIs of Twitter and Facebook to retrieve the *platform descriptions* in order to "learn" how to use the platform-specific identifiers to retrieve representations of David's user accounts. We present descriptions for the Twitter and Facebook platforms in Chapter 8.

7.2.1.2 Manipulation of digital artifacts via representations

Per C2, *digital artifacts* must be manipulated via representations exchanged between components. Representations are serializations of the current or intended states of *digital artifacts* and their purpose is to hide implementation details, such as platform-specific data types and models, behind a set of standard media types (see Section 2.1.1 for more details).

This interface constraint emphasizes a *minimal requirement* for any interface that is to be integrated into the SWoT into a useful manner: the interface should at least produce some representations of resources that can be mapped to *digital artifact descriptions* (see Section 7.1.4).

7.2.1.3 Self-descriptive messages

Per C3, messages exchanged between SWoT components have to be self-descriptive, which implies that SWoT components must be able to reliably process messages using only standardized knowledge, such as standard methods (e.g., the methods defined by HTTP 1.1 [Fielding 2014c] or CoAP [Shelby 2014a]) and standard media types[11]. It is, therefore, necessary to standardize the processing of domain-specific *digital artifact* representations, which can be achieved by:

- defining new media types to fit SWoT-specific requirements;
- extending standard media types via semantic information.

Defining new media types can be particularly useful in the context of the WoT as a means to optimize the production and consumption of resource representations for resource-constrained devices. Examples of data formats and media types designed for constrained devices include, among others, the Concise Binary Object Representation [Bormann 2013] data format and associated media type, and ongoing work on media types for sensor measurements[12]. We leave it as future work to further investigate the development of SWoT-specific media types for highly resource-constrained devices (see Chapter 10).

Throughout the rest of this dissertation, we take the second approach, which may be preferable for less constrained devices and general purpose applications: we use standard media types for RDF serialization formats, in particular Turtle [Beckett 2008], and the STN ontology. The advantage of decoupling the interpretation of representation semantics from media types is that clients could potentially adapt to new vocabularies more easily than to new media types.

This interface constraint raises two challenges for the integration of existing platforms into the SWoT (cf. discussion in Section 2.1.2). First, it is not uncommon for existing HTTP-based APIs to use the protocol in a non-standard manner, such as deleting resources via `HTTP POST`. Second, in the absence of standard media types for their particular application domains (e.g., social applications), most existing Web platforms expose platform-specific data models serialized using standard data interchange formats, such as JSON [Bray 2014]. These factors lead to interface heterogeneity and, in order to interact with each platform, clients must

[11]A complete list of registered media types is available at: http://www.iana.org/assignments/media-types/media-types.xhtml, Accessed: 25.11.2015.

[12]https://tools.ietf.org/html/draft-jennings-core-senml-02, Accessed: 22.11.2015.

hard-code knowledge provided by out-of-band documentation. As a practical solution to address these issues and integrate existing platforms into the SWoT, platform authorities or third parties can use the *STN ontology* (see Section 7.1.1) to publish *platform descriptions* (see Section 7.1.3). In Chapter 8, for instance, we present *platform descriptions* for Facebook, Twitter, SoundCloud, and Dweet.io.

7.2.1.4 Hypermedia-driven interaction

Per C4, the interaction between a client and an STN platform must be *driven by hypermedia*. That is to say, in any given application state, the client should be able to choose from a selection of states provided by the STN platform. Furthermore, the instructions to transition to a new state should be provided via hypermedia (see Section 2.1.1 for further clarifications). Hypermedia-driven interaction is central to decoupling clients from STN platforms.

In a SWoT application, a client can transition to new application states by:

- following references encoded in representations of *digital artifacts* (e.g., hyperlinks, David's Twitter account description in Listing 7.1);
- performing *operations* (see Definition 6.1.11) on *digital artifacts*;
- receiving notifications from an *STN platform* when the state of a *digital artifact* has changed, if such an interaction mechanism is implemented (e.g., via the CoAP `Observe` option [Hartke 2015]).

Given one or more entry points into a hypermedia-driven SWoT, such as the URI of David's STN Box (see examples in Section 7.1), a *social thing* should be able to autonomously participate in a continuously evolving ecosystem, without hard-coding references that could break at any time. This interface constraint is thus closely related to the discovery of *entities* (see Definition 6.1.1) in the SWoT. Our proposal relies on several elements to enhance *hypermedia-driven interaction* and *discoverability*:

- the *social connectivity principle* (see Foundational Principle 2) enforces relations among *entities* in the SWoT, which are reified in the digital world via references across *digital artifacts*, such as *user accounts* or *SWoT profiles*;
- *SWoT profiles* enable the discovery of *user accounts* held by *agents* across various *STN platforms*, and thus serve as *convergence points* for digital STNs that may otherwise be disconnected from one another (e.g., David's *SWoT profile* in Listing 7.1 enables the discovery of the digital STNs hosted by David's STN Box, Twitter, and Facebook);
- *hosting relations* (see Definition 6.1.24) encoded in *digital artifact descriptions* (see Section 7.1.4) via the `stn:hostedBy` property (see Section 7.1.1.1) enable the discovery of *platform descriptions* (see Section 7.1.3).

Most existing Web platforms expose non-hypermedia APIs (see Section 2.1.2). In the absence of hypermedia, interaction with the APIs is driven by out-of-band information provided as human-readable documentation. Consequently, clients must hard-code the knowledge required to transition between application states, such as URIs or URI templates (see also our discussion of interface constraint C3). *Platform descriptions* provide a means to represent this knowledge in a reusable, machine-readable format, and *hosting relations* provide a means to make *platform descriptions* discoverable, therefore essentially bringing in-band all the out-of-band information that clients need in order to autonomously retrieve and manipulate *digital artifacts* in the SWoT regardless of the underlying hosting platforms.

It is worth to note that third parties can create and publish *platform descriptions* to extend the SWoT to existing platforms, and non-hypermedia STN platforms in general, with minimal effort and without requiring "buy-in" from platform authorities, which is an important factor for the successful development of the SWoT. Such *platform descriptions*, however, are static and have to be constantly maintained in order to reflect the evolution of the described STN platforms, an issue that can be alleviated to some extent via API versioning. In other words, *platform descriptions* enable clients to "learn" on-the-fly how to interface with non-hypermedia STN platforms, however, once they do, they become *tightly coupled* to the described interfaces: if the *platform descriptions* become obsolete, the clients break. Developers of STN platforms that aim for a stronger integration into the SWoT should provide hypermedia APIs.

Hypermedia controls

A distinctive characteristic of a hypermedia API is that it conveys to clients the next reachable states in an application and how to make transitions to those states. The information is encoded in controls that are included in representations transmitted to clients and processed according to specified media types. For instance, HTML forms instruct Web browsers how to collect user input and transmit it to an origin server. Therefore, hypermedia controls extend standard interaction protocols, such as HTTP [Fielding 2014c] or CoAP [Shelby 2014a], with information meant to further specify the interaction between components in a dynamic fashion. This is an important factor to minimize the coupling between clients and STN platforms. However, the disadvantage of embedding hypermedia controls in representations exchanged between components is decreased network efficiency. Network efficiency is especially important in the context of the IoT.

Per our discussion of constraint C3 (see Section 7.2.1.3), we leave it as future work to develop SWoT-specific media types optimized for resource-constrained devices. Nevertheless, hypermedia APIs can already use the STN ontology to include *operation descriptions* (see Section 7.1.3.1) in RDF representations transmitted to clients. A more detailed discussion on the development of hypermedia APIs is available in [Webber 2010].

Linked Data

Hypermedia controls trade network efficiency for loose coupling. An approach to alleviate the former is to further standardize interaction in the SWoT. That is to say, clients and STN platforms can hard-code SWoT-specific conventions for interacting with other components via standard interaction protocols, and leave it to hypermedia controls to deliver any platform- and application-specific details.

Based on the suggestions we presented thus far, it is worth to note at this point that the world-wide STG is, in fact, represented in the SWoT as *linked data* (see Section 2.3.3). The *Linked Data Platform (LDP)* [Speicher 2015] already provides a standard set of conventions for reading and writing linked data on the Web via HTTP. Furthermore, the LDP enforces connectivity, for instance, by imposing that conforming *LDP servers* must automatically manage *containment* and *membership* relations between *LDP containers* and the *LDP resources* they contain (see [Speicher 2015] for details).

We suggest that the LDP can provide a standard-compliant foundation for interaction between clients and STN platforms. The LDP can be further extended with SWoT-specific conventions, such as imposing on STN platforms to automatically manage *hosting relations* whenever a *digital artifact* is created in a *digital artifact container* (see Section 7.1.3.2). We leave it as future work to further investigate this possibility (see Chapter 10), in particular in the context of resource-constrained devices, which raises new questions. For instance, the LDP is currently based exclusively on HTTP, however, CoAP is a cornerstone protocol for the WoT (see Section 2.4). CoAP supports a subset of the features provided by HTTP, which includes content negotiation and conditional requests [Shelby 2014a]. Many of the core LDP conventions may thus be directly applicable to constrained environments via HTTP-CoAP translation [Shelby 2014a], while other aspects remain to be investigated.

In the following sections, we further discuss our suggestions for integrating heterogeneous STN platforms into a hypermedia-driven environment for the SWoT. We classify heterogeneous interfaces based on their mode of interaction with clients by answering the question: who performs *operations* (see Definition 6.1.11) on *digital artifacts* (see Definition 6.1.22)?

7.2.2 Control-driven interfaces

We say that an interface of an STN platform is *control-driven* if it is defined in terms of *operations* (see Definition 6.1.11) that are invoked remotely by clients and executed on the STN platform.

It is worth to note that many existing Web APIs are control-driven (see Section 2.1.2). For an illustration, "following" (i.e., creating a relation to) a Twitter account via the Twitter Public API v1.1 can be performed by issuing the following HTTP request:[13]

[13]Example provided by the Twitter Public API v1.1 documentation:

```
POST https://api.twitter.com/1.1/friendships/create.json?user_id=1401881
```

In this example, the request URI encodes the operation to be performed (i.e., *create friendship*) and a required parameter whose value is a platform-specific identifier of the targeted user. This approach of transferring information over the Web is also referred to as *URI tunneling* [Webber 2010]. If the operation is successful, the API returns a JSON-based platform-specific representation of the followed Twitter account. Therefore, conceptually mapping the *CreateRelationTo* operation defined by our *digital STN model* to Twitter's API is straightforward (cf. formal definition in Section 6.3.2):

- the *platform* parameter is determined by the *authority component* of the URI [Berners-Lee 2005];
- the *performing agent* parameter is determined by means of *HTTP authentication* [Fielding 2014a];
- the *targeted user account* is determined via an explicit input parameter;
- the output provides a representation that can be mapped to a *digital artifact description* (see Listing 7.5).

The operation is invoked remotely by the client and performed by the Twitter platform. It is worth to note that HTTP is used here as a transport protocol, and not as an application protocol. Deleting a relation to a Twitter account is performed via a similar `HTTP POST` request by replacing "create" with "destroy" in the request URI. Therefore, we conclude that this example, and URI tunneling in general, breaks uniformity constraints C1, C3, and C4 (see Section 7.2.1).

Generalizing from our above example, control-driven non-hypermedia STN platforms can be integrated into the SWoT by mapping the *operations* defined by our *digital STN model* (see Section 6.3) to the ones supported by the platform's interface via *platform descriptions* (see Section 7.1.3), which can be advertised to clients by means of *hosting relations* (see Section 7.2.1.4). In other words, clients are provided with machine-readable *service contracts* that encode the knowledge necessary to interface with the platforms. These service contracts, however, are *static*, which we have already discussed in Section 7.2.1.4. We discuss further the advantages and disadvantages of this integration strategy in Section 7.3.

Developers of control-driven STN platforms that wish to achieve a stronger integration into the SWoT can use the STN ontology and follow our suggestions in Section 7.2.1 to provide a hypermedia API.

7.2.3 Data-driven interfaces

We say that an interface of an STN platform is *data-driven* if it is defined in terms of *digital artifacts* exchanged with clients by means of methods and data formats

https://dev.twitter.com/rest/reference/post/friendships/create, Accessed: 22.11.2015.

with a shared understanding between clients and the STN platform. In data-driven interactions, the *operations* defined by our *digital STN model* (see Section 6.3.2) are performed on the client side, and the resulting states of *digital artifacts* are then transferred to STN platforms.

Per our discussion in Section 2.1.2, many existing Web APIs implement a Create, Read, Update, Delete (CRUD) interface via a subset of the HTTP verbs (e.g., `GET`, `POST`, `PUT`, `DELETE`), also referred to as *CRUD Web services* [Webber 2010]. These non-hypermedia services use HTTP as an application protocol, however, they generally violate uniformity constraints C1, C3, and C4 (see Section 7.2.1): they use platform-specific identifiers, platform-specific data models serialized using general data interchange formats (e.g., JSON [Bray 2014]), and are driven by out-of-band information. It is worth to note, in fact, that existing CRUD Web services exhibit behavior that is similar to non-hypermedia control-driven interfaces: the *operation* to be performed is encoded via an *(HTTP method, URI)* tuple that is hard-coded into the client, and the *operation*'s parameters are encoded within the transferred representation, in most cases in terms of key-value pairs. Therefore, existing CRUD Web services can be integrated into a hypermedia-driven SWoT in a similar manner to control-driven interfaces, that is by providing *platform descriptions* (see Section 7.1.3) to translate the *operations* defined by our *digital STN model* to the service. The main integration shortcoming, however, remains that clients have to rely on static service contracts to interact with the platforms.

To achieve a stronger integration into the SWoT, developers of data-driven STN platforms can follow our suggestions in Section 7.2.1 to expose hypermedia interfaces, for instance, by providing clients with data-guided controls or an LDP-compliant interface.

7.2.4 Mixed interfaces

Following our discussion in the previous sections, it is worth to note that existing Web APIs are highly heterogeneous and it is not always straightforward to classify them as *control-* or *data-driven*. In most cases, however, Web APIs behave in a manner that is more similar to control-driven interaction, whereas the REST architectural style is centered around data-driven interaction. Furthermore, even though non-hypermedia APIs are not aligned with the REST architectural style, they are simple and intuitive to most developers, which is an important argument for their success on the Web. It is also worth to note that public Web APIs tend to have a low change frequency and to use API versioning, which alleviates the integration problems related to their evolution over time.

Given the above arguments and that we provide solutions to integrate control-driven non-hypermedia APIs into a hypermedia-driven environment for the SWoT, a claim that we demonstrate in Chapter 8, it is worth to bring into discussion the possibility of exposing *mixed interfaces* for STN platforms. In some use cases, mixed interfaces could be useful to leverage the benefits of both worlds: provide a baseline data-driven interface (e.g., a linked data API) that follows our *digital*

STN model to ensure a deep integration into the SWoT, and extend this interface with simple control-driven interfaces that can be easily understood by developers and can, for instance, extend our *digital STN model* with new *digital artifacts* and artifact-specific *operations*. As a practical example, a few existing WoT platforms already allow developers to dynamically extend platform capabilities by publishing application-specific Web APIs to be hosted by the platforms (see Section 3.1).

7.3 A Five-level Integration Strategy for STN Platforms

We apply the elements and suggestions introduced in the previous sections to propose a progressive, five-level strategy for the development of STN platforms and their integration into the SWoT. Each level increasingly restricts the design and implementation autonomy of STN platforms to the benefit of achieving greater alignment with the SWoT, which in turn ensures greater interoperability with SWoT clients. For instance, most existing social platforms can be integrated into the SWoT as *Level 1 STN platforms* with minimal effort and without requiring the support of platform authorities, however, their utility to *social things* would most likely be limited. *Level 5 STN platforms* must conform to several design and implementation choices, however, they are also most useful to *social things* and necessary building blocks of the envisioned IoT ecosystem.

It is worth to note that we generally use the term *STN platform* to refer to platforms that interconnect people and things in network-like structures that can be represented as *STNs* (see Chapter 6). Per Foundational Principle 2 (social connectivity), having explicit typed relations among people and things is central to our vision and an important factor to enhance discoverability in the SWoT (see Section 7.2.1.4). Nevertheless, the SWoT can also benefit from platforms that enable interaction among people and/or things without explicitly representing the relations among them. This is the case, for instance, of most existing WoT platforms.

7.3.1 Level 1: Publish a platform description

The first step to integrate any platform into the SWoT is to provide a *platform description* (see Section 7.1.3) that enables software clients to reliably interface with the platform. Any heterogeneous platform that provides a valid *platform description* is a *Level 1 STN platform*.

The most straightforward approach to publish a *platform description* on the Web is as a standalone *STN description document* (see Section 7.1.3), which can be created and advertised by platform authorities or trusted third parties. A *platform description* can also be distributed across multiple interlinked documents reachable from the platform's *STN description document*, which may be useful, for instance, to optimize network usage or to isolate components of an interface that tend to change more frequently than others.

We suggest to publish *STN description documents* via a standardized *well-known URI* [Nottingham 2010], such as `/.well-known/stn`. This suggestion helps to prevent URI collisions and provides a means to avoid imposing unnecessary constraints on the URI structure of STN platforms (cf. best practices for URI design and ownership [Nottingham 2014]). Clients can also hard-code the standardized URI to easily test if a random platform they encounter is an STN platform or not.

The main advantage of this integration strategy is that third parties can extend the SWoT to existing platforms with minimal effort and without requiring "buy-in" from platform authorities, which is an important factor for the successful development of the SWoT. Another advantage is that *STN description documents* can be discovered, accessed and used in a simple manner. The documents can be cached (e.g., locally, close to the edge of the network) to improve network efficiency.

An important disadvantage of relying on static *platform descriptions*, as already discussed in Section 7.2.1.4, is that they have to be constantly maintained in order to reflect the evolution of the described platforms. In the case of existing public Web APIs, however, as noted in Section 7.2.4, the change frequency is generally low and this integration issue can also be mitigated further via API versioning.

Another drawback of relying solely on *STN description documents* is that they can become significantly large, and thus less suitable for resource-constrained devices. Furthermore, this integration strategy relies on clients to integrate data from heterogeneous sources, which can be a costly task. These problems can be mitigated via *intermediary components* (i.e., *proxies*) that encapsulate the heterogeneous interfaces of one or more STN platforms behind a uniform interface optimized for resource-constrained devices.

7.3.2 Level 2: Enable social things as first-class citizens

A core tenet of our vision is that *social things* are enabled as *first-class citizens* of the SWoT (see Section 5.3.1). A *Level 2 STN platform* is any *Level 1 STN platform* that exposes via its API all the basic operations required to access the platform's main features, such as exchanging messages or managing social relations with other users. Social things are, therefore, full-fledged users of Level 2 STN platforms. Most existing WoT platforms fit this description and can be integrated into the SWoT as Level 2 STN platforms. It is worth to note, however, that WoT platforms do not typically expose explicit relations among things (see Chapter 3).

Depending on the degree of alignment between a platform's model and our *digital STN model* (see Section 6.3), social things may have limited access to platform features. In addition, the data integration step can also still limit the utility of Level 2 STN platforms. These integration issues can be alleviated by extending our *digital STN model* and the *STN ontology* (see Section 7.1.1) with domain- and platform-specific models and vocabularies.

7.3.3 Level 3: Produce STN-compliant representations

A *Level 3 STN platform* is any *Level 2 STN platform* that produces representations of *social artifacts* in conformance with our *digital STN model* (see Section 6.3.3), using terms defined by the STN ontology, and standard media types.

Avoiding the cumbersome task of integrating data from heterogeneous sources greatly simplifies the integration process. For SWoT clients, it enables them to directly consume representations produced by STN platforms, which decreases integration costs and avoids intermediary components or loss of data. For STN platforms, it increases the audience in terms of potential clients. It also diminishes tight coupling by significantly reducing the out-of-band information required to interface with the platform, and thus also reducing the size of *STN description documents*.

It is worth to note that, by conforming to the digital STN model, it is expected that Level 3 STN platforms provide explicit representations of relations among their users. This constraint enforces hypermedia-driven interaction and discoverability in the SWoT.

7.3.4 Level 4: Expose a uniform API

Producing representations that any SWoT client can reliably interpret is already an important step towards exposing a *uniform interface*. Going further, a *Level 4 STN platform* is any *Level 3 STN platform* that exposes an API conforming to the uniformity constraints in Section 7.2.1.

Uniform interfaces bring the important benefit of loose coupling, which simplifies the development of SWoT clients and ensures that STN platforms provide services that are compatible with and useful to a larger number of SWoT clients. Uniform interfaces are essential to the development of a long-lived IoT ecosystem in which components can be deployed and can evolve independently from one another.

Per our discussion in Section 7.2.1.4, it is worth to note that hypermedia APIs, and in particular linked data, seem to be a promising approach for extending the SWoT to resource-constrained devices.

7.3.5 Level 5: Make the platform open

The last step towards achieving a greater integration into the SWoT is support for *openness*. A *Level 5 STN platform* is any *Level 4 STN platform* that supports cross-platform participation and open standards and mechanisms for uniquely identifying SWoT clients.

Support for cross-platform participation implies that identifiable *agents* in the SWoT should be able to interact with other identifiable *agents* without being confined to the STN platforms they use. A cross-platform feature that is important to enhance discoverability in the SWoT is to allow *agents* to create relations to/from *entities* in the ecosystem, regardless of their hosting platforms. In our scenarios in Section 5.1, David and his friends own STN Boxes, and the relations among them span across their STN Boxes.

Securely and uniquely identifying *agents* globally is an important problem to be addressed, which we leave as future work (see Chapter 10). It is worth to note, however, that practical solutions to this problem are already within reach.

HTTP benefits from many security-related features, and there are ongoing efforts for the standardization of authentication and authorization mechanisms for constrained networks[14]. We suggest to investigate the use of open standards and decentralized single sign-on systems, such as OpenID Connect [Sakimura 2014] and WebID [Sambra 2015a], with certificate-based authentication for *social things* whenever appropriate.

7.4 Summary

In this chapter, we addressed Research Question 3, that is *how to enable things to transcend Web silos.* Our approach is to integrate *heterogeneous STN platforms* into a *hypermedia-driven environment* for the SWoT. To this purpose, we applied the REST architectural style (per Foundational Principle 1) and the digital STN model (see Section 6.3) to provide solutions for achieving uniform interfaces for heterogeneous STN platforms, and for enhancing hypermedia-driven interaction and discoverability across STN platforms.

In Section 7.1, we introduced a semantic description framework that we use to create descriptions of digital STNs and their hosting platforms. *Platform descriptions* (see Section 7.1.3) are central to our approach for dealing with platform heterogeneity: they enable platform authorities and third parties to represent the knowledge that is required to interface with heterogeneous STN platforms in machine-readable format, knowledge that is otherwise provided as out-of-band human-readable documentation (see Section 2.1.2). SWoT clients can then reliably process *platform descriptions* to "learn" on-the-fly how to interface with heterogeneous platforms. We applied our semantic description framework in Section 7.2, where we discussed solutions to create hypermedia APIs for STN platforms and to integrate heterogeneous, non-hypermedia APIs into a hypermedia-driven SWoT. In Section 7.3, we proposed a five-level strategy for the development of STN platforms and their integration into the SWoT that enables a progressive alignment with our vision.

It is worth to note that our approach for integrating existing platforms into the SWoT does not impact existing APIs and does not require "buy-in" from platform authorities. We demonstrate this claim, and our approach to create a hypermedia-driven SWoT, in the next chapter, where we deploy a world-wide socio-technical graph that extends to multiple existing platforms, such as Facebook and Twitter.

[14]Such as the efforts of the IETF Authentication and Authorization for Constrained Environments (ACE) working group: https://datatracker.ietf.org/wg/ace/documents/, Accessed: 27.11.2015.

Part III

Experience and Validation

Chapter 8

Deploying a World-Wide Socio-technical Graph

Contents

In Part II of this dissertation, we presented our vision and approach to bring about an open and self-governed IoT ecosystem of people and things, which we call the *Social Web of Things (SWoT)*. Our proposal emphasizes *heterogeneity*, *discoverability*, and *flexible interaction* in the envisioned IoT ecosystem. We begin the evaluation of our work by focusing on the former characteristic. We discuss the other two in Chapter 9.

The backbone of the SWoT is a global environment sustained by *heterogeneous platforms*. In Chapter 6, we defined a model for digital *socio-technical networks (STNs)*, and in Chapter 7 we applied this model and the REST architectural style (per Foundational Principle 1) to propose solutions to create a *hypermedia-driven environment* for the SWoT. The defining characteristic of this environment is that it hosts a world-wide *socio-technical graph (STG)* (see Definition 6.1.10) distributed across *heterogeneous STN platforms* that *social things* (see Definition 6.1.7) can navigate and manipulate in a uniform fashion.

In this chapter, we validate our proposal by deploying an STG distributed across multiple existing social and WoT platforms, and across multiple instances of a *Level 5 STN platform* (see Section 7.3). In Section 8.1, we discuss the integration into the SWoT of several widely used social platforms, namely *Facebook*, *SoundCloud* and *Twitter*, and a WoT platform, that is *Dweet.io*. In Section 8.2, we present our implementation of a *Level 5 STN platform*, which we call *ThingsNet*. In Section 8.3, we deploy an STG distributed across all these platforms, and show how a Web application that can interpret *platform descriptions* (see Section 7.1.3) enables human users to seamlessly navigate and manipulate the deployed STG stating from a single entry point in the SWoT. This application, which we call an *STN browser*, is thus able to "learn" on-the-fly how to interface with the platforms underlying the deployed STG.

8.1 Integrating Existing Platforms into the SWoT

In this section, we discuss the integration of *Facebook*, *SoundCloud*, *Twitter* and *Dweet.io* into the SWoT. Based on our integration strategy in Section 7.3, Facebook can be integrated into the SWoT as a *Level 1 STN platform* (i.e., social things *are not* first-class citizens), and the other three platforms as *Level 2 STN platforms* (i.e., social things *are* first-class citizens). We discuss this further in the following sections. Facebook, SoundCloud and Twitter can contribute to the world-wide STG with their social graphs, whereas Dweet.io does not feature explicit relations between things.

For each platform, we structure our discussion as follows:

- **Platform overview**: we present briefly the major features of the platform that are relevant in the context of the SWoT following the dimensions defined in Section 6.1, that is the *digital*, *social*, *spatial* and *normative* dimensions;

- **API overview**: we provide an overview of the platform's API, discuss to what extent *social things* can access and use platform features, and provide examples of API calls that can be used to implement *operations* defined by our *digital STN model* (see Section 6.3.2);

- **STN platform description**: we discuss the platform's *STN description document* (see Section 7.1.3) that we create using the *STN ontology* (see Section 7.1.1).

The complete *STN description documents* for the platforms presented in this section are available in Appendix A.

After discussing each platform individually, in Section 8.1.5 we conclude with an analysis of the extent to which these platforms can be "interweaved" into a world-wide STG.

8.1.1 Facebook

Facebook[1] is an online social networking service that enables its users to create and maintain online relations with one other. As of September 30, 2015, Facebook has a staggering 1.55 billion monthly active users.[2]

8.1.1.1 Platform overview

Facebook is a social platform *designed for people*, and has a strong policy against user accounts not associated with real people, user accounts that provide false personal information, or people with multiple user accounts.[3] Organizations and other entities may have a presence on Facebook via Facebook Pages. As such, *social things* cannot be first-class citizens on Facebook, and thus Facebook can be integrated into the SWoT as a *Level 1 STN platform* (see Section 7.3). That is to say, given a valid *platform description* (see Section 7.1.3), *social things* can use the Facebook platform primarily to read data from its rich social graph.

The digital dimension

Facebook *user accounts* carry rich information about their holders, which may include general information (e.g., name, website, contact details), albums of personal photographs and videos, or various things the user likes, such as books, movies, or music. Many real-world *entities* have digital counterparts in the form of Facebook Pages, which may fall in various categories (e.g., TV shows, sports teams, restaurants). *Places* have digital counterparts via Facebook Places, which can thus be represented as *digital places* (see Definition 6.1.29). Other types of *digital artifacts* include Facebook Groups, which can be represented as *digital groups* (see Definition 6.1.27), private or public messages, which can be represented as *digital messages* (see Definition 6.1.28).

The social dimension

Facebook users can establish bidirectional "friendship" relations with one another, which can be represented as two unidirectional *social relations* (see Definition 6.1.13), via a request/response interaction model. Some Facebook relations carry more *meaning* than the standard friendship relations, such as family relations (e.g., mother, brother). Users can interact via messages (e.g., private messages, public posts and comments, tagging), and they can reinforce (i.e., "like") and disseminate content created by other users. The latter features, and interaction with content in general, are not currently covered by our *digital STN model.* Users can also create, join, leave or invite other users to Facebook Groups, which may be used, for instance, to enable and concentrate communication with Facebook users sharing a common interest. These features can be represented via the *digital STN model.*

[1]http://www.facebook.com/

[2]http://newsroom.fb.com/company-info/, Accessed: 28.11.2015.

[3]https://www.facebook.com/legal/terms, Accessed: 28.11.2015.

The spatial dimension

Public messages can carry geospatial information via linking to a Facebook Place. It is worth to note that Facebook does not feature a specific *operation type* (see Definition 6.1.12) to "check into" a *digital place* (see Section 6.3.3), however, the location of a user may be inferred based on the content he or she publishes.

The normative dimension

Interactions on Facebook, from navigating the social graph to disseminating information, are governed by complex privacy settings managed by users.

8.1.1.2 API overview

The Facebook platform provides a non-hypermedia API, that is the Facebook Graph API v2.4[4]. It is worth to note that platform resources are, in fact, assigned URIs of the form `/{api_version}/{node_id}`, however, returned representations include only platform-specific identifiers (i..e, the `node_id` parameter). The API produces JSON-based [Bray 2014] representations using platform-specific data models. These observations can be noted in the API example in Listing 8.1. Therefore, Facebook's API breaks uniformity constraints C1, C3 and C4 in Section 7.2.1.

As noted in the previous section, Facebook's policies enforce the preservation of a social network composed exclusively of people described by accurate information. Furthermore, Facebook's API provides a rich variety of endpoints to read data from Facebook's social graph, whereas support to write data to the social graph is much more limited. For instance, several endpoints expose the various facets of the data associated with a user account (i..e, a Facebook profile), however, a third party application can post public messages only *on behalf* of a Facebook user and it cannot, for instance, send private messages or manage social relations with other users.

Listing 8.1 shows a Facebook-specific implementation of the *GetOutgoingRelations operation type* defined in Section 6.3.2. One of the particularities of Facebook's API is that, for instance, requests to retrieve representations of *user accounts* rely on query parameters to specify the fields to be returned within the representations (cf. Listing 8.1). The returned fields are subject to privacy policies and typically require explicit permission from users.

```
GET /v2.4/1550387481863557/friends?fields=id,name,website HTTP/1.1
Host: graph.facebook.com
Authorization: Bearer CAAByOV4ZA5O...mHIZA78gZDZD

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
ETag: "babd881296d1ab6c09115a9e0c699fa5ea2e803a"
Facebook-API-Version: v2.4
```

[4] https://developers.facebook.com/docs/graph-api, Accessed: 28.11.2015.

```
Content-Length: 511

{
   "data": [
      {
         "id": "295402373983233",
         "name": "Jennifer Amedjdahfiif Schrockson"
      },
      {
         "id": "1475781352713478",
         "name": "John Amhddhhbbihd Fergieman"
      }
   ],
   "paging": {
      "next": "https://graph.facebook.com/v2.4/1550387481863557/friends
          ?fields=id,name,website&limit=25&offset=25&__after_id=enc_Ad
          ..XX"
   },
   "summary": {
      "total_count": 2
   }
}
```

Listing 8.1: The Facebook friends of a user identified by `{user_id}` are retrieved via an HTTP `GET` request to `/v2.4/{user_id}/friends?fields={account_fields}`. The request header includes an OAuth access token [Hardt 2012] that authorizes the request. Access to most user account fields requires explicit user-granted permission, such as the `website` field in this example.

8.1.1.3 STN description document

The STN description document we have created for Facebook is available in Appendix A. An extract from this document is shown in Listing 8.2. Facebook's STN platform description includes multiple agent operations (see Section 6.3.2), such as retrieving the outgoing and incoming social relations of a user account. Given that on Facebook social relations are bidirectional, both of these operations are, in fact, implemented in the same manner. It is worth to note that in our description, we only request the `id`, `name`, and `website` fields (cf. Listing 8.2). The latter field may be used, for instance, to retrieve *digital artifacts* outside of the social network. In Section 8.3, we use the `website` field to advertise the *SWoT profiles* (see Section 7.1.2) of Facebook users. More detailed representations of user accounts may be requested via the *GetUserAccount operation type* (see Section 6.3.3.1).

```
@base <http://www.facebook.com/> .

<#platform>
    a stn:Platform ;
    stn:name "Facebook" ;
    stn-http:baseURL <https://graph.facebook.com/v2.4/> ;
    stn-http:supportsAuth stn-http:OAuth ;
    stn-http:consumes stn-http:JSON ;
    stn-http:produces stn-http:JSON ;
    stn-ops:supports <#getAccount> ,
        <#getOutConnections> ,
    <#getInConnections> ,
    ...
    <#getGroupFeed> .

<#getOutConnections>
    a stn-ops:GetOutgoingRelations ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/friends?fields=id,name,website" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput <#fbFriendsJSONMapping> .
```

Listing 8.2: Retrieving the friends of a Facebook user is performed via an authorized HTTP request and requires one path parameter as input, i.e. the identifier of the targeted user account.

The output mapping for retrieving the Facebook friends of a user, shown in Listing 8.3, extracts the JSON array of user account representations from the payload returned by the Facebook API (cf. Listing 8.1) and maps each element of the array to an RDF representation of an `stn:UserAccount` (see Section 7.1.4).

```
<#fbFriendsJSONMapping>
    a stn-http:JSONArray ;
    stn-http:key "data" ;
    stn-http:arrayOf <#fbAccountJSONMapping> .

<#fbAccountJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:UserAccount ;
    stn-http:contains [
            a stn-http:Mapping ;
            stn-http:key "id" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-http:contains [
            a stn-http:Mapping ;
            stn-http:key "name" ;
            stn-http:STNTerm stn:name ;
        ] ;
    stn-http:contains [
            a stn-http:Mapping ;
            stn-http:key "website" ;
            stn-http:STNTerm stn:swotProfile ;
        ] .
```

Listing 8.3: This mapping enables a social thing to extract RDF data from the JSON payload in Listing 8.1. In this description, the `website` field, if any, is mapped to the URL of the user's SWoT profile (see Section 8.1.5.3 for details).

8.1.2 SoundCloud

SoundCloud[5] is an online audio distribution platform. Among other features, SoundCloud enables its registered users to record and upload audio content, or promote and disseminate audio content created by other users. SoundCloud has over 250 million monthly active users.[6]

8.1.2.1 Platform overview

SoundCloud's terms of use[7] do not explicitly restrict registered users to people, and in fact many organizations and companies use SoundCloud for hosting "podcasts"[8]. The platform's API supports most of the operations available to regular service users, and therefore SoundCloud can be integrated into the SWoT as a *Level 2 STN platform* (see Section 7.3).

The digital dimension

SoundCloud *user accounts* carry basic information about their holders (e.g., name, description, city, website). Other *digital artifacts* defined by our digital STN model (see Section 6.3.3) that can be found on SoundCloud are *digital groups* and comments, that is *messages* with reference to a *digital artifact*. Domain-specific *digital artifacts* include tracks and playlists. Registered users can interact with *digital artifacts* in various ways, for instance by joining or leaving groups, liking and reposting tracks.

The social dimension

SoundCloud users can connect to other users via unidirectional relations, which can be mapped to *social relations* (cf. Definition 6.1.13). Users can interact directly with one another via private messages, or indirectly by means of comments attached to tracks.

[5]http://www.soundcloud.com/

[6]http://techcrunch.com/2013/10/29/soundcloud-now-reaches-250-million-listeners-in-its-quest-to-become-the-audio-platform-of-the-web/, Accessed: 28.11.2015.

[7]https://soundcloud.com/terms-of-use/, Accessed: 09.09.2015.

[8]Such as: https://soundcloud.com/product-hunt/, Accessed: 09.09.2015.

The spatial dimension

SoundCloud does not currently feature digital artifacts with spatial characteristics.

The normative dimension

Interactions on SoundCloud are not generally restricted or sanctioned by any norms. For an exception, when publishing a track, users can choose to disable comments for that track.

8.1.2.2 API overview

The SoundCloud API is resource-oriented and provides endpoints for most *digital artifacts* hosted by the platform, with one exception being private messages.[9] Read operations generally require only the identifier of the SoundCloud application using the API, and thus we consider them to be publicly available. Creating or modifying resources is generally performed on behalf of a registered user and requires authorization via OAuth [Hardt 2012].

```
GET /users/169098327/followings?oauth_token=1-14...f4c7 HTTP/1.1
Host: api.soundcloud.com

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
ETag: "81344ff2b737132ca84a63bfccbd8c7f"
Content-Length: 569

[
    {
        "id": 88985927,
        "kind": "user",
        (...),
        "description": "Product Hunt Radio (PHR) is for (..) best new
            products, every day.",
        "city": "San Francisco",
        (...),
        "website": null,
        "website_title": null,
        (...)
    }
]
```

Listing 8.4: The SoundCloud user accounts followed by a user identified by `{user_id}` are retrieved via an HTTP `GET` request to `/{user_id}/followings`. The OAuth access token [Hardt 2012] is passed as a query parameter.

Listing 8.4 shows how the *GetOutgoingRelations operation type* (see Section 6.3.2) can be implemented via the SoundCloud API. The HTTP request is similar with the

[9]https://developers.soundcloud.com/docs/api/reference/, Accessed: 09.09.2015.

one issued to the Facebook Graph API (cf. Listing 8.1), however on SoundCloud resources are generally public and their representations encode their entire state. The response returned by the platform encloses in its body a JSON array [Bray 2014] of user account representations that follow a platform-specific user account model. Unlike the Facebook Graph API, pagination has to be requested explicitly via a query parameter when using the SoundCloud API. It is worth to note that *digital artifacts* on SoundCloud are identified by both platform-specific identifiers and URIs, and the URIs are also included in their representations.

8.1.2.3 STN description document

The *STN description document* we have created for SoundCloud is similar to the one created for Facebook and is available in Appendix A. SoundCloud's STN platform description includes most *operations* defined for reading and manipulating *social relations*, user accounts, and groups (see Section 6.3).

8.1.3 Twitter

Twitter[10] is an online social networking service that enables its users to send and read messages of at most 140 characters, called "tweets". As of September 30, 2015, Twitter has 320 million monthly active users.[11]

8.1.3.1 Platform overview

Similar to Facebook and SoundCloud, Twitter is a service primarily designed for people, however, Twitter defines its users explicitly as "anyone or anything"[12]. The platform's API supports most of the operations available to regular service users, and therefore Twitter can be integrated into the SWoT as a *Level 2 STN platform* (see Section 7.3).

The digital dimension

Twitter *user accounts* can provide basic information about their holders, such as a short biography, their location and website. Things can hold and use Twitter *user accounts*. *Places* may also have a digital counterpart on Twitter.[13] Other *digital artifacts* include private and public *messages* (i.e., tweets), and lists of users. Registered users can favorite, retweet and reply to tweets, and they can create or subscribe to user lists.

[10]http://www.twitter.com/
[11]https://about.twitter.com/company, Accessed: 28.11.2015.
[12]https://dev.twitter.com/overview/api/users, Accessed: 06.09.2015.
[13]https://dev.twitter.com/overview/api/places, Accessed: 09.09.2015.

The social dimension

Twitter users can connect to other users via unidirectional relations, which can be mapped to *social relations* (cf. Definition 6.1.13). Users can choose to have "protected" accounts in order to approve who can follow their user accounts. Users can interact directly with one another via direct messages, or indirectly via tweets.

The spatial dimension

Tweets can carry geospatial information by linking to the identifier of a location, which may be represented as a *digital place.*

The normative dimension

Most information on Twitter is public, with the exception of users choosing protected accounts. It is worth to note that the Twitter API provides endpoints for retrieving the platform's privacy policy and terms of services as unstructured text. Retrieving the rate limit status of an application is also available via the API.

8.1.3.2 API overview

The Twitter Public API v1.1 supports most actions that are available to Twitter users. Most resources on Twitter are public, however all HTTP requests have to be authenticated such that they identify the requesting application. Any write operations are generally performed on behalf of a Twitter user and require authorization via OAuth 1.0a [Hammer-Lahav 2010].

Listing 8.4 shows how the *GetOutgoingRelations operation type* (see Section 6.3.2) can be implemented via Twitter's API. Unlike the similar HTTP requests for Facebook (cf. Listing 8.1) and SoundCloud (cf. Listing 8.4), the platform-specific identifier is passed as a query parameter. *User accounts*, in particular, have two identifiers, namely the `id` field, which holds an integer, and the `screen_name` field, which holds a string (cf. Listing 8.5). Representations enclosed in responses follow platform-specific models and include only platform-specific identifiers of *digital artifacts.*

```
GET /1.1/friends/list.json?cursor=-1&screen_name=twitterapi HTTP/1.1
Host: api.twitter.com

HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
ETag: "babd881296d1ab6c09115a9e0c699fa5ea2e803a"
Content-Length: 511

{
  "previous_cursor": 0,
  "previous_cursor_str": "0",
  "next_cursor": 1333504313713126852,
  "users": [
    {
      (...),
      "id_str": "657693",
      (...),
      "url": "http://afroginthevalley.com/",
      (...),
      "followers_count": 4993,
      "protected": false ,
      (...),
      "description": "Developer Advocate at Twitter. (...)",
      (...),
      "friends_count": 2743,
      "following": true,
      "screen_name": "froginthevalley"
    },
    (...)
  ],
  "next_cursor_str": "1333504313713126852"
}
```

Listing 8.5: The list of users followed by a Twitter account identified by `{user_id}` is retrieved via an HTTP `GET` to `/1.1/friends.json?screen_name={user_id}`. The user account fields to be included in the response have to be requested explicitly as query parameters. Access to most fields requires explicit user permission, such as the `website` field.

8.1.3.3 STN platform description

The *STN description document* we have created for Twitter is similar to the one created for Facebook and is available in Appendix A. The mapping for extracting an RDF representation of a user account from a JSON-based representation of a Twitter account was already presented in Section 7.1.4. Twitter's *STN platform description* includes most operations defined for reading and manipulating *social relations*, *user accounts*, and *messages* (see Section 6.3).

8.1.4 Dweet.io

Dweet.io[14] is a cloud-based IoT platform that enables things to publish and consume data using self-assigned URIs.

8.1.4.1 Platform overview

Dweet.io enables *things* to publish data objects, also referred to as "dweets", to a self-assigned URI of the form `https://dweet.io/dweet/for/{thing_name}`, where `thing_name` should be a unique identifier. Data may be transmitted via query parameters or in a JSON payload. At most 500 dweets are stored for a period of 24 hours for any Dweet.io URI. Things can openly access Dweet.io URIs both to publish and to consume data. Dweet.io also provides a payed feature (i.e., "locks") that

[14]http://www.dweet.io/

enables developers to restrict access to their URIs and receive rule-based notifications when dweets are being published, for instance when the JSON object contains a field whose value is over a given threshold.

Things are *first-class entities* on Dweet.io and thus the platform can be integrated into the SWoT as a *Level 2 STN platform* (see Section 7.3). It is worth to note that Dweet.io does not feature explicit relations among *things*, however this is not a requirement for *Level 2 STN platforms*.

8.1.4.2 API overview

Dweet.io provides a Humanized Web API[15] and exposes endpoints for publishing and retrieving dweets. Listing 8.6 shows an HTTP request for publishing a dweet and the response returned by the API. Listing 8.7 shows an HTTP request for retrieving the dweets published at a given URI.

```
POST /dweet/for/example.org HTTP/1.1
Host: dweet.io
Content-Type: application/json
Cache-Control: no-cache

{ "hello": "world" }


HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 146

{
  "this": "succeeded",
  "by": "dweeting",
  "the": "dweet",
  "with": {
    "thing": "example.org",
    "created": "2015-09-07T09:21:59.764Z",
    "content": {
      "hello": "world"
    }
  }
}
```

Listing 8.6: Publishing a dweet for a thing with the identifier `example.org`. A JSON representation of the dweet to be published is included in the request body.

```
GET /get/dweets/for/example.org HTTP/1.1
Host: dweet.io

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 148
```

[15]http://github.com/jheising/HAPI, Accessed: 07.09.2015.

```
{
    "this": "succeeded",
    "by": "getting",
    "the": "dweets",
    "with": [
        {
            "thing": "example.org",
            "created": "2015-09-07T09:21:59.764Z",
            "content": {
                "hello": "world"
            }
        }
    ]
}
```

Listing 8.7: Retrieving available dweets published via the URI of a thing with the identifier `example.org`.

8.1.4.3 STN description document

The *STN description document* we have created for Dweet.io is available in Appendix A. An extract from this document is shown in Listing 8.8, which includes a description for retrieving the dweets posted by a thing. Listing 8.9 shows a description for a *CreateUserAccount operation type* (see Section 6.3.3.1), which can be used for “creating” a user account on Dweet.io. Even though Dweet.io does not, in fact, require registration, this operation can be simulated for consistency with the *digital STN model*, which enables *social things* to interact with the platform using known patterns and in-band information.

```
<#platform>
    a stn:Platform ;
    stn:name "Dweet.io" ;
    stn-http:baseURL <https://dweet.io> ;
    stn-ops:consumes stn-http:JSON ;
    stn-ops:produces stn-http:JSON ;
    stn-ops:supports <#createAccount> ,
        <#postDweet> ,
        <#getDweets> .

<#getDweets>
    a stn-ops:GetUserAccountFeed;
    stn-ops:implementedAs [
            a stn-http:STNRequest ;
            http:methodName "GET" ;
            http:requestURI "/get/dweets/for/:accountId" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key ":accountId" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput [
            a stn-http:JSONArray ;
            stn-http:key "with" ;
            stn-http:arrayOf <#dweetJSONMapping> ;
        ] .

<#dweetJSONMapping>
    a stn-http:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:DataObject ;
    stn-http:contains [
            a stn-http:Mapping ;
            stn-http:key "created" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-http:contains [
            a stn-http:Mapping ;
            stn-http:key "thing" ;
            stn-http:STNTerm stn:createdBy ;
        ] ;
    stn-http:contains [
            a stn-http:Mapping ;
            stn-http:key "content" ;
            stn-http:STNTerm stn:data ;
        ] .
```

Listing 8.8: Extract from Dweet.io's STN platform description. Retrieving the dweets published by a *social thing* requires the platform-specific identifier of its "user account" in order to retrieve its user account feed. The output of the operation is a *JSON array* of *data objects*.

```
<#createAccount>
    a stn-ops:CreateUserAccount ;
    stn-ops:implementedAs [
            a stn-http:STNRequest ;
            http:methodName "GET" ;
            http:requestURI "/dweet/for/:agentURI" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-http:AgentURI ;
            stn-http:key ":agentURI" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput [
            a stn-http:JSONRepresentation ;
            stn-ops:mediaType stn-http:JSON ;
            stn-ops:entityType stn:UserAccount ;
            stn-http:key "with" ;
            stn-http:contains [
                    a stn-http:Mapping ;
                    stn-http:key "thing" ;
                    stn-http:STNTerm stn:createdBy;
                    ] ;
          ] .
```

Listing 8.9: Operation description for the *CreateUserAccount operation type* (see Section 6.3.3) as implemented by the Dweet.io platform.

8.1.5 Discussion

We analyze the degree to which the platforms presented in this section can be interweaved into the SWoT by looking at three criteria:

- what are the *operation types* (see Section 6.3.2) that *social things* could use on each platform;
- what are any *major platform features* that are *not covered* by the *digital STN model* (see Section 6.3);
- openness and navigability, that is to what extent can *social things* access and "crawl" the platforms, transition seamlessly to other platforms, or perform any cross-platform operations.

8.1.5.1 Platform support for STN operations

A summary of the *operation types* supported by the platforms discussed in this section is shown in Table 8.1.

Table 8.1: Types of *operations* supported by the APIs of Facebook, SoundCloud, Twitter, and Dweet.io. We summarize *operations types* in terms of the Create, Read, Update, Delete (CRUD) methods supported by each type of *digital artifact*.

Platform	User Accounts	Social Relations		Groups	Direct Messages	Feeds	
		Type	Access			Type	Access
Facebook	R, U	bidirectional	R	R	R	Home, User, Group	C, R, D
SoundCloud	R	unidirectional	C, R, D	R	-	-	-
Twitter	R, U	unidirectional	C, R, D	-	C, R, D	Home, User	C, R, D
Dweet.io	C, R	-	-	-	-	User	C, R

Social things can perform *read operations* for all *digital artifacts* defined by our *digital STN model* and supported by each platform. Support for read operations has high utility particularly in the case of Facebook, which has a staggering 1.55 billion monthly active users (as of September 30, 2015)[16], Twitter, which has 320 million monthly active users (as of September 30, 2015)[17], and SoundCloud, which has over 250 million monthly active users[18]. On these platforms, *social things* can retrieve

[16]http://newsroom.fb.com/company-info/, Accessed: 28.11.2015.

[17]https://about.twitter.com/company, Accessed: 28.11.2015.

[18]http://techcrunch.com/2013/10/29/soundcloud-now-reaches-250-million-listeners-in-its-quest-to-become-the-audio-platform-of-the-web/, Accessed: 07.09.2015.

representations of *user accounts*, crawl *social graphs*, and access user-generated content.

Support for *write operations* is somewhat more limited on Facebook: *social things* would only be able to post and delete public messages on behalf of a human user. On SoundCloud and Twitter, *social things* would be able to perform most of the *operations* that are available to human users (cf. Table 8.1). They can, for instance, create and delete *social relations*. On Dweet.io, social things are able to create *user accounts*, which is a simulated *operation* (see Section 8.1.4), and to post/read messages to/from *user account feeds*.

8.1.5.2 Platform features not covered by STN operations

The Facebook Graph API provides access to a rich amount of information about its users, such as movies and books they like. SoundCloud also provides access to useful domain-specific features, such as audio streaming. The *digital STN model* (see Section 6.3) is a general model whose purpose is to provide an extensible backbone for the SWoT. *Social things* would require domain-specific extensions in order to access and use these features. Similarly, Facebook, Twitter and SoundCloud provide features for promoting or disseminating to social media items (e.g., like, share, repost, retweet). An extension that would enable agents to interact with social media is discussed as future work in Section 10.3. Another feature, which is important for practical reasons, that we leave as future work is *pagination*.

8.1.5.3 Openness and navigability

The Facebook Graph API v2.4 and the SoundCloud API use OAuth 2.0 [Hardt 2012], and the Twitter Public API v1.1 uses OAuth 1.0a [Hammer-Lahav 2010] to authorize HTTP requests. This implies that in order to access these platforms *social things* would have to be configured against each platform, which hinders openness and navigability. If able to access the platforms, *social things* would then generally be able to crawl their social graphs. On Facebook, however, the possible crawl depth and the information extracted in the process would depend heavily on the permissions granted to social things and the various privacy settings of Facebook users. On all these platforms, crawling, as well as other operations, would also be subject to rate limiting policies. On Twitter, social things would be able to retrieve their rate limits dynamically via the platform's API, given they support an extension that enables them to do so (see Section 10.3). On SoundCloud, rate limits are out-of-band information provided in the API documentation. On Facebook, rate limits are not currently presented explicitly to developers.

Access to locked Dweet.io URIs is currently performed by means of fixed keys that social things would have to possess.[19] Unlocked Dweet.io URIs are accessed openly. Dweet.io does not currently feature explicit relations between things, and thus it is not possible to crawl the platform to discover other *social things*. In

[19]https://dweet.io/locks, Accessed: 28.11.2015.

Section 8.3, we show how we can address this limitation by integrating Dweet.io in the world-wide STG and leveraging the social graphs of other platforms.

Social things can transition seamlessly to any of these platforms from *SWoT profiles* (see Section 7.1.2) or STN platforms that support *cross-platform relations*. Transitioning from any of these platforms to other platforms is not straightforward, however, various conventions can help to achieve this purpose. For instance, users of social platforms could set their `websites`, a field used by most social platforms, to the URLs of their *SWoT profiles*. It is worth to note that such conventions do not have to be hard-coded into *social things*, they can be included in the mappings used to extract RDF representations of *user accounts* via the `stn:swotProfile` property (see Listing 8.3).

8.1.5.4 Conclusions

In summary, *social things* can use *STN platform descriptions* (see Section 7.1.3) to crawl Facebook, SoundCloud, and Twitter. On Facebook and SoundCloud they can also retrieve representations of groups and group members, which may help during crawling. Facebook and Twitter provide various feeds that social things could consume. Domain-specific extension can further enhance the information extracted from these platforms.

Based on our integration strategy in (see Section 7.3), Facebook can be integrated into the SWoT as a *Level 1 STN platform*, which means that *social things* are not first-class entities, however, they can use the platform as a data source. Twitter, SoundCloud, and Dweet.io can be integrated into the SWoT as *Level 2 STN platforms*. On Twitter and SoundCloud, *social things* can hold *user accounts*, build and maintain social graphs. On Twitter, they can also interact with people via direct messages and tweets. *Social things* can use Dweet.io to publish and consume JSON data objects. Given that Dweet.io does not provide a graph of relations that could be crawled, *social things* have to be provided with the URIs of other social things in order to consume any data they may publish.

The platforms presented in this section, and other similar platforms, can be interweaved into the SWoT by means of *STN platform descriptions*, *SWoT profiles* and STN platforms that support *cross-platform relations*. We present one such platform in the following section.

8.2 ThingsNet: a Level 5 STN Platform

In the previous section, we discussed the integration of several existing platforms into the SWoT as either *Level 1* or *Level 2 STN platforms*. In this section, we present our implementation of a *Level 5 STN platform*, which we call *ThingsNet*.

ThingsNet enables *agents* to create and manage *social relations* (see Definition 6.1.13) with other *agents* in the SWoT, and to interact with one another by means of *messages*. ThingsNet is a *Level 5 STN platform* (see Section 7.3) because it allows *social things* to participate in the STN as first-class citizens, it produces

STN-compliant representations of *digital artifacts*, it provides a uniform API, and it is an open platform. ThingsNet relies on WebID [Sambra 2015a] to uniquely identify *agents* in the SWoT, and it supports cross-platform participation (i) by allowing *social relations* to *agents* registered on other STN platforms and (ii) by allowing identifiable *agents* registered on other STN platforms to send messages to ThingsNet users.

8.2.1 Design and implementation

In a rough description, ThingsNet is essentially a repository of *digital artifacts*. All *digital artifacts* are identified by URIs and their states are fully represented in RDF using the STN ontology (see Section 7.1.1). The implementation is written in Scala [Odersky 2004] using the Play Web framework[20] and banana-rdf[21].

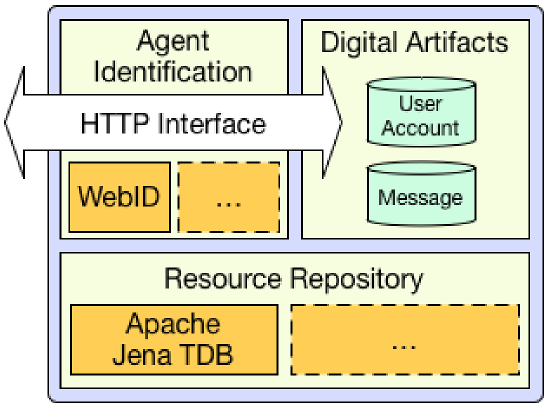


Figure 8.1: Architectural elements of the ThingsNet platform. The states of digital artifacts are fully represented in RDF and stored as named graphs. Representations of digital artifacts are retrieved via an HTTP-based interface. Some requests (e.g., retrieving the messages received by a user) require authentication via the WebID authentication protocol.

The architectural elements of ThingsNet are depicted in Figure 8.1. Our prototype implementation simulates the WebID authentication protocol [Sambra 2015a] to uniquely identify requesting *agents*, and uses Apache Jena TDB[22] to store and query RDF representations of *digital artifacts*. The platform has a modular design such that its components can be easily substituted or complemented (cf. Figure 8.1).

ThingsNet currently features two types of *digital artifacts*, that is *user accounts* and *messages* (cf. Figure 8.1). *User accounts* carry basic information about their holders, such as their WebIDs, names and descriptions to be displayed within the STN. In addition, *user accounts* held by *social things* include the WebIDs of the

[20]https://www.playframework.com/, Accessed: 28.11.2015.
[21]https://github.com/banana-rdf/banana-rdf/, Accessed: 28.11.2015.
[22]https://jena.apache.org/documentation/tdb/index.html, Accessed: 28.11.2015.

social things' owners. *Messages* can have one or more recipients, and can contain a subject and a body.

Our current implementation does not feature a front-end for human users. Nevertheless, people can access ThingsNet by using any SWoT-compliant software client, such as the *STN browser* we present in Section 8.3.2.

8.2.2 API overview

ThingsNet provides an HTTP-based hypermedia API that conforms to the *uniformity constraints* in Section 7.2.1: *digital artifacts* are identified via URIs, clients interact with the API by exchanging representations of *digital artifacts*, representations are produced using the Turtle RDF serialization format [Prud'hommeaux 2014] and the STN ontology, and interaction with the API is driven by hypermedia.

ThingsNet uses standard HTTP and provides an *STN description document* that encodes all platform-specific knowledge required to interface with the platform. The document is published via the `/.well-known/stn` URI and identifies the platform via the `/.well-known/stn#platform` hash URI (see Section 7.3). All representations of *digital artifacts* produced by ThingsNet include its platform URI via a *hosting relation* (see Definition 6.1.24). ThingsNet's complete STN description document is available in Appendix A.

For illustrative purposes, the description of the *CreateUserAccount operation type* (see Section 6.3.3) is shown in Listing 8.10. Following our formal definition in Section 6.3.3, the *performing agent* is identified via the WebID authentication protocol and the *platform parameter* is implicit. In addition, ThingsNet requires a number of other parameters (cf. Listing 8.10): the *social thing's class*[23], which may be the class of *social things* denoted by `stn:SocialThing` or any of its subclasses, an URI identifying the social thing's *owner*, and a *name* to be displayed within the STN. Optionally, a *description* may also be provided. This *operation type* returns a Turtle representation of a *user account*.

```
@prefix format: <http://www.w3.org/ns/formats/> .

<#platform>
    a stn:STNPlatform ;
    stn:name "ThingsNet" ;
    stn-http:baseURL <http://localhost:9000> ;
    stn-http:supportsAuth stn-http:WebID ;
    stn-http:consumes format:Turtle ;
    stn-http:produces format:Turtle ;
    stn-ops:supports <#createAccount> ,
        <#getAccount> ,
         ...
        <#deleteMessage> .

<#createAccount>
    a stn-ops:CreateUserAccount ;
```

[23]Our prototype implementation assumes that only *social things* register via ThingsNet's API.

```
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "POST" ;
            http:requestURI "/users/" ;
        ] ;
    stn-ops:hasRequiredInput [ a stn-ops:SocialThingClass ] ;
    stn-ops:hasRequiredInput [ a stn-ops:Owner ] ;
    stn-ops:hasRequiredInput [ a stn-ops:DisplayedName ] ;
    stn-ops:hasInput         [ a stn-ops:Description ] ;
    stn-ops:hasOutput [
            a stn-http:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:entityType stn:UserAccount ;
        ] .
```

Listing 8.10: An extract from ThingsNet's *STN description document.* The document includes general information about the platform (e.g., name) and its API (e.g., supported authentication protocol and media types), and descriptions for each of the *operation types* supported by the platform. This listing shows the *operation description* for the *CreateUserAccount* operation type as implemented by the ThingsNet platform.

Following the operation description in Listing 8.10, David's social TV in Section 5.1.1 can thus register to ThingsNet using a WebID provided by its manufacturer, the WebID of its owner (i.e., David), a pre-configured social thing class, and a user-configured name. We report on our implementation of this application scenario in Section 9.2. The social TV interprets the operation description to construct a Turtle representation of the user account to be created, which includes all the required parameters, and issues an `HTTP POST` request to the `/users/` endpoint, which is shown in Listing 8.11. ThingsNet responds with a `201 Created` status code. The response payload contains a Turtle representation of the created *user account*, which includes the user account's URI generated by the platform and a hosting relation (cf. Listing 8.10).

```
POST /users HTTP/1.1
Host: localhost:9000
Content-Type: text/turtle
X-WebID: http://api.mymanufacturer.com/tvs/874...260#thing

@prefix : <http://purl.org/stn/core#> .
@prefix ex: <http://www.example.com#> .

<> a :UserAccount ;
    :name "David's Social TV" ;
    :description "A TV with a twist!" ;
    :heldBy <http://api.mymanufacturer.com/tvs/874...260#thing> .

<http://api.mymanufacturer.com/tvs/874...260#thing>
    a ex:SocialTV ;
    :ownedBy ex:David .
```

```
HTTP/1.1 201 Created
Content-Type: text/turtle

@prefix stn: <http://purl.org/stn/core#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://www.example.com#> .

<http://localhost:9000/users/f8d...a16>
    a stn:UserAccount ;
    stn:description
        "A TV with a twist!" ;
    stn:heldBy
        <http://api.mymanufacturer.com/tvs/874...260#thing> ;
    stn:hostedBy
        <http://localhost:9000/.well-known/stn#platform> ;
    stn:name
        "David's Social TV" .

<http://api.mymanufacturer.com/tvs/874...260#thing>
    a ex:SocialTV ;
    stn:ownedBy ex:David .
```

Listing 8.11: A request-response interaction for creating a user account on ThingsNet. The request payload includes a Turtle representation of the user account to be created. The response payload includes a Turtle representation of the created user account, which contains the platform-generated URI of the user account and a hosting relation.

Our prototype implementation does not, in fact, implement the WebID authentication protocol [Sambra 2015a], however, it simulates it by passing the WebID of the requesting agent in a custom HTTP header field, that is `X-WebID` (cf. Listing 8.11). The prototype implementation returns a `401 Unauthorized` response whenever authentication data is missing or the requesting agent does not have access to perform the intended operation, such as retrieving a registered user's messages.

8.2.3 Discussion

By conforming to all the requirements of our five-level integration strategy in Section 7.3, the ThingsNet platform becomes a core component of the SWoT infrastructure, which in turn could increases its audience in terms of clients that can consume its services. In the next section, we report on using our prototype implementation to "glue together" the deployed STG.

Per our suggestion in Section 7.2.1, the Linked Data Platform (LDP) [Speicher 2015] could provide a standard-compliant foundation for interaction between SWoT components. Our design and implementation choices for the development of ThingsNet were, at many times, guided by the LDP. Our prototype implementation, however, does not fully conform to the normative requirements of the LDP. Furthermore, some of the SWoT-specific features provided by ThingsNet are implemented in a manner that is not in-line with the interaction patterns promoted by the LDP. For instance, our prototype implements the *GetOugoingRelations* operation type (see Section 6.3.2) via an `HTTP GET` to the `/connections/out` endpoint with a query parameter that specifies the URI of the targeted user account. The advantage of this approach is that it simplifies development and testing. A disadvantage, however, is that it pollutes the URI space in the SWoT.

Outgoing relations are, in fact, a subset of the set of triples that describes a user account (cf. Definition 7.1.3). An approach more in-line with the interaction patterns promoted by the LDP would be to request the representation of the target user account, however, by also providing to the STN platform a SWoT-specific hint about the desired representation (i.e., only outgoing relations) via the HTTP `Prefer` header field [Snell 2014]. We leave it as future work to define a linked data protocol for STNs (see Chapter 10.3).

Our experience with the development of ThingsNet confirms, unsurprisingly, that RDF provides a data model well-suited for representing the world-wide STG. Named graphs allow for a natural partitioning of the STG across digital artifacts, which can then be easily merged together.

8.3 Deployment of a World-Wide Socio-technical Graph

In this section, we validate our approach by creating a SWoT environment that is *driven by hypermedia* and *sustained by heterogeneous platforms*. To this purpose, we use the platforms discussed in Section 8.1 and ThingsNet (see Section 8.2) to deploy a SWoT environment for the scenario introduced in Section 5.1.1 ("The Social TV"). We present our deployment scenario in Section 8.3.1. In Section 8.3.2, we present a Web application that people can use to navigate and participate into the SWoT, which we call an *STN browser*. In Section 8.3.3, we illustrate how a user can use the STN browser to navigate and manipulate the deployed STG, starting only from an entry URI and regardless of the underlying heterogeneous platforms.

8.3.1 Deployment scenario

Our deployment scenario is depicted in Figure 8.2 (cf. scenario in Section 5.1.1). We create a distributed STG using Facebook, Twitter, SoundCloud, and ThingsNet. We use five instances of ThingsNet, which run locally on various ports, to implement the STN Boxes of David and his friends. The three social platforms are closed, whereas the STN Boxes are open platforms (cf. Figure 8.2). David and his friends own social TVs, which use Dweet.io to publish data, such as movie ratings collected from their owners (cf. Figure 8.2).

As illustrated in Figure 8.2, David has a single *social relation* (see Definition 6.1.13) in his home STN, via his *user account*, to a friend on a different STN Box. David, however, is also registered on each of the three social platforms, and on each platform he is connected, in a platform-specific manner, to a friend that also has an STN Box and *owns* a social TV. The social graphs we use on Facebook,

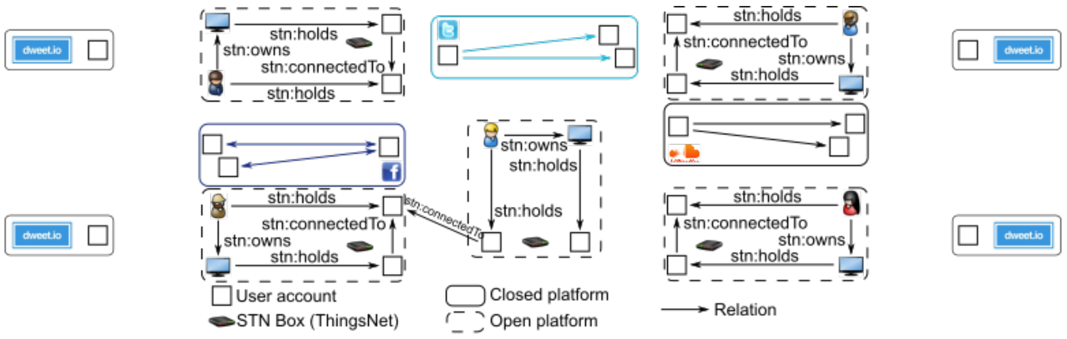


Figure 8.2: Deployment scenario: David owns an STN Box, which hosts his home STN. In addition, David holds multiple user accounts on various other platforms. Some of David's friends on those platforms also own STN Boxes. The social TVs use Dweet.io to publish data.

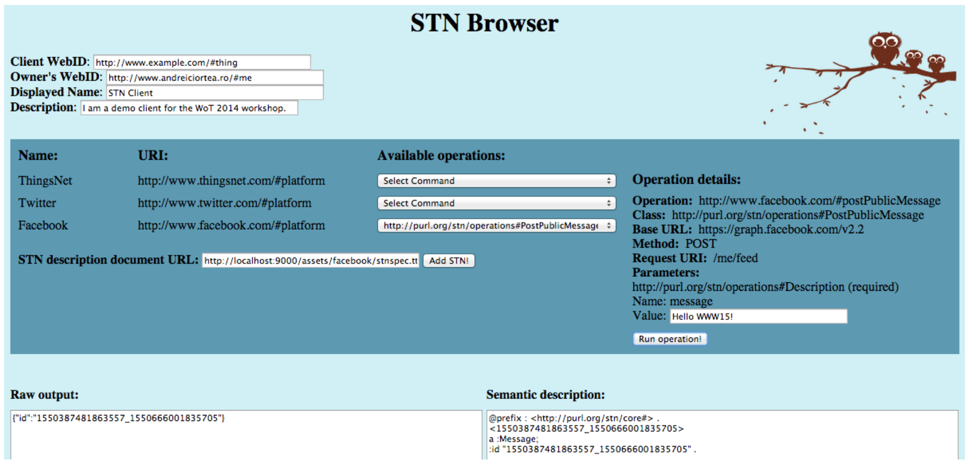
Twitter and SoundCloud have been created manually via multiple development *user accounts*. Facebook provides developers with a harness to test their applications via *test users*. Nevertheless, the user accounts of *test users* function similarly to regular user accounts (e.g., they provide the same fields for personal information and require the same permissions to access them). For Twitter and SoundCloud, we use regular user accounts.

David, his friends, and his friends' social TVs have *SWoT profiles* (see Section 7.1.2) that are hosted by their STN Boxes and include their *WebID profiles* [Sambra 2015a] as well. We assume these *SWoT profiles* have been created when the *agents* registered to the STN Boxes. We assume David's social TV has a *SWoT profile* hosted by its manufacturer, which includes its *WebID profile* as well. David and his friends advertise the URLs of their *SWoT profiles* as their personal websites on all the social platforms they use (see discussion in Section 8.1.5.3).

8.3.2 A browser for STNs

We have developed a Web application, which we call an *STN browser*, that enables people to navigate and participate into the SWoT. The *STN browser* acts as a uniform user interface for heterogeneous STN platforms: it retrieves and interprets *STN description documents* (see Section 7.1.3), based on which it displays controls that can be used to perform *operations* (see Section 6.3.2) supported by the described platforms.

The user interface of the *STN browser* is shown in Figure 8.3. The *STN browser* has a number of pre-configured parameters, which are displayed in the top left corner, such as its WebID, the WebID of its owner (e.g., to enact *social things* for testing purposes), a name or a description to be displayed within an STN. The panel in the middle left of the user interface provides controls that allow a user to load an *STN description document* from a given location. When a new *platform description* is loaded, the described platform is added to the list of available *STN platforms* and



STN Browser

Client WebID: http://www.example.com/#thing
Owner's WebID: http://www.andreiciortea.ro/#me
Displayed Name: STN Client
Description: I am a demo client for the WoT 2014 workshop.

Name:	**URI:**	**Available operations:**
ThingsNet	http://www.thingsnet.com/#platform	Select Command
Twitter	http://www.twitter.com/#platform	Select Command
Facebook	http://www.facebook.com/#platform	http://purl.org/stn/operations#PostPublicMessage

STN description document URL: http://localhost:9000/assets/facebook/stnspec.tt

Operation details:
Operation: http://www.facebook.com/#postPublicMessage
Class: http://purl.org/stn/operations#PostPublicMessage
Base URL: https://graph.facebook.com/v2.2
Method: POST
Request URI: /me/feed
Parameters:
http://purl.org/stn/operations#Description (required)
Name: message
Value: Hello WWW15!

Raw output:

```
{"id":"1550387481863557_1550666001835705"}
```

Semantic description:

```
@prefix : <http://purl.org/stn/core#> .
<1550387481863557_1550666001835705>
a :Message;
:id "1550387481863557_1550666001835705" .
```

Figure 8.3: The STN browser displays the result of performing an agent operation in two areas: the area on the left shows the raw body of the response returned by the origin server, and the one on the right shows the semantic information it can extract from that response. This image shows the result of posting a status update on Facebook.

the *operations* it supports are added to an associated drop-down list (cf. Figure 8.3).

When a user selects an *operation* to be performed on one of the available *STN platforms*, the controls required to perform the operation are dynamically displayed in the control panel to the right. This panel includes a number of text fields that describe the operation and an input field for each of the parameters that are *required* to perform the operation (cf. Figure 8.3). The *STN browser* auto-completes any parameters that are pre-configured, if any, by matching required parameters against its list of pre-configured parameters based on their declared types (see *STN-Operations* in Section 7.1.1.2). In Figure 8.3, for instance, the *STN browser* displays controls for posting a public message on Facebook. The operation class is denoted by `stn-ops:PostPublicMessage` and it has a required input parameter, whose type is denoted by `stn-ops:Description`.

The output of an *operation* is displayed in the bottom panel of the user interface in two areas (cf. Figure 8.3). The area on the left displays the raw body of the HTTP response received from an STN platform. The area on the right displays the RDF data that the STN browser is able to extract from that response. In Figure 8.3, the area on the left displays the raw JSON payload received from the Facebook platform after posting a public message (i.e., a Facebook status), which returns only a platform-specific identifier of the created message. The STN browser extracts the identifier from this representation and displays a Turtle representation of the created message in the area on the right. A user can use this platform-specific identifier to perform an `stn-ops:GetMessage` operation and retrieve a complete representation of the message, for instance one that would also include the message body.

Like ThingsNet, the STN browser is written in Scala using the Play Web framework and banana-rdf. The front-end is written in HTML and JavaScript, and uses Asynchronous JavaScript and XML (AJAX) to interface with the application's back-end and display information in a dynamic manner. The back-end provides a server-side proxy to construct and perform all necessary HTTP requests. The requests are built automatically as specified by the *platform descriptions*. It is worth to note, however, that all social platforms use OAuth-based authorization for HTTP requests, which implies that the requests have to be configured for authorization against each social platform (see Section 1.8 on interoperability of RFC 6749). We use Scribe[24], an OAuth library, to sign HTTP requests for Facebook, SoundCloud and Twitter. We use unlocked URIs for Dweet.io (see Section 8.1.4) and we simulate the WebID authentication protocol when interfacing with ThingsNet nodes (see Section 8.2.2). Our current implementation of the STN browser supports Turtle and JSON media types.

8.3.3 Discussion: browsing the Social Web of Things

The *STN description documents* of the various platforms in our deployment scenario are used to achieve a *uniform interface* (see Section 7.2) based on our *digital STN model* (see Section 6.3) and using the STN ontology (see Section 7.1.1). This uniform interface decouples the *STN browser* from the heterogeneous platforms underlying the deployed STG. The interweaving of the otherwise isolated STGs of each platform is then achieved via *cross-platform relations*, *SWoT profiles*, and *hosting relations* (per our discussion in Section 7.2.1.4). These elements enable the navigation of the deployed STG *across* its underlying platforms, given only an entry point in the form of a URI. We illustrate this behavior in what follows.

8.3.3.1 Hypermedia-driven interaction

Returning to our scenario depicted in Figure 8.2, we choose our entry point into the SWoT to be David's *SWoT profile*, which is retrieved from his STN Box. In our deployment scenario, David's *SWoT profile* is part of the description of David's *user account* on his STN Box, which is shown in Listing 8.12, and we assume it was generated when David registered to the STN Box. The *SWoT profile*, however, could also be hosted as a standalone document, similar to a *FOAF profile* [Brickley 2014]. The STN Box is implemented as an instance of ThingsNet that runs locally on port `9000` (cf. Listing 8.12).

David's *SWoT profile* contains descriptions of all *user accounts* held by David on the platforms he uses in this scenario (cf. Listing 8.12). These descriptions include *hosting relations*, denoted via `stn:hostedBy` properties, that point to the heterogeneous platforms underlying the STG (cf. Listing 8.12). In our scenario implementation, the *STN description documents* for Facebook, Twitter and SoundCloud are published by David's STN Box (cf. Listing 8.12). The user account

[24]https://github.com/fernandezpablo85/scribe-java, Accessed: 28.11.2015.. As a technical detail, it is worth to note that Scribe is a Java library. Scala, however, runs on the Java virtual machine (JVM), and invoking Java code from Scala is performed seamlessly.

held by David on his STN Box, which is a *hypermedia-driven platform*, is uniquely identified via a URI (cf. Listing 8.12). Dereferencing this URI retrieves the Turtle representation in Listing 8.12.

```
<http://localhost:9000/users/e75...8f3#David>
    a stn:Person ;
    stn:name "David" ;
    stn:holds <http://localhost:9000/users/e75...8f3> ;
    stn:holds [
        a stn:UserAccount ;
        stn:hostedBy <http://localhost:9000/assets/stnspecs/twitter.ttl
            #platform> ;
        stn:id "swotdev" ;
    ] ;
    stn:holds [
        a stn:UserAccount ;
        stn:hostedBy <http://localhost:9000/assets/stnspecs/facebook.
            ttl#platform> ;
        stn:id "1550387481863557" ;
    ] ;
    stn:holds [
        a stn:UserAccount ;
        stn:hostedBy <http://localhost:9000/assets/stnspecs/soundcloud#
            platform> ;
        stn:id "swotdev" ;
    ] ;
    cert:key [ a cert:RSAPublicKey ;
        cert:modulus "cb24ed...1391a1"^^xsd:hexBinary ;
        cert:exponent 65537 ;
    ] .

<http://localhost:9000/users/e75...8f3>
    a stn:UserAccount ;
    stn:description ( "Doe. David Doe." ) ;
    stn:heldBy <http://localhost:9000/users/e75...8f3#David> ;
    stn:hostedBy <http://localhost:9000/.well-known/stn#platform> ;
    stn:name "David Doe" ;
    stn:connectedTo <http://localhost:9001/users/c2e...f16> .
```

Listing 8.12: A representation of David's user account on his STN Box, which includes David's SWoT profile. The SWoT profile identifies David as a *person*, and describes the *user accounts* he holds on his STN Box, Twitter, Facebook and SoundCloud. The SWoT profile includes David's public key, which could be used, for instance, in the WebID authentication protocol [Sambra 2015a].

Having *discovered* the URIs of the social platforms in our deployments scenario via David's *SWoT profile*, we can now use the *STN browser* to load each *STN description document* and perform *operations*, for instance to retrieve complete representations of David's user accounts by means of a *GetUserAccount* operation or to retrieve the outgoing social relations established via his user accounts by means of a *GetOutgoingRelations* operation (see Section 6.3.3), which is used to further navigate the STG. In Section 8.1, we presented platform-specific descriptions and implementations for the *GetOutgoingRelations operation type* for each of the social platforms in our deployment scenario.

Operations that can be used via the *STN browser* to manipulate the deployed STG include creating and deleting social relations, creating and deleting messages. Some social platforms support other types of operations as well (cf. Table 8.1).

8.3.3.2 Conclusions

The user can follow steps similar to the ones discussed so far to continue navigating the deployed STG. Figure 8.4 depicts the client's view of this STG. From a SWoT client's perspective, the STG is *uniform* and can be *manipulated in a uniform fashion*, even though the STG is, in fact, distributed across heterogeneous platforms using platform-specific data models and APIs.

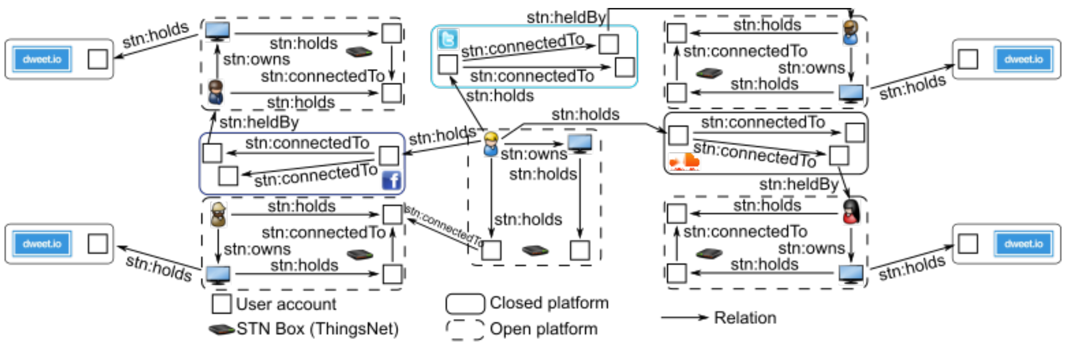


Figure 8.4: An overview of the semantic representation that the STN browser is able to build for the social graphs deployed in our scenario, effectively creating a distributed socio-technical graph.

We say that the deployed STG is *world-wide* because the client is agnostic to the underlying platforms. It is also worth to emphasize that we have successfully integrated into the deployed STG two of the largest social platforms available at the moment of writing this dissertation, that is Facebook (see Section 8.1.1) and Twitter (see Section 8.1.3).

We say that the deployed SWoT environment is *hypermedia-driven* because a SWoT client, such as our *STN browser*, can navigate and manipulate the world-wide STG starting from a single entry point, such as the URI of David's *SWoT profile*.

Dweet.io does not feature relations among its users (see Section 8.1.4). An *important consequence* of integrating this platform into the deployed SWoT environment is that the social TVs' data streams published via the platform are now *discoverable*. By crawling the STG, the user can reach the Dweet.io user accounts at the edge of the deployed STG. In Section 9.2, in which we discuss our implementation of the "The Social TV" scenario in Section 5.1.1, David's social TV implements this behavior to aggregate movie ratings published by the social TVs of David's friends. Without this enabled discoverability, David's social TV would have to *hard-code* references to the Dweet.io user accounts.

The instances of ThingsNet deployed in this scenario play an essential role: they enhance hypermedia-interaction via *cross-platform relations*. The *SWoT profiles* of the *agents* involved in this scenario could have also been hosted as standalone documents on regular Web servers. Nevertheless, ThingsNet offers a preview of what the SWoT could look like if it is to be sustained by *Level 5 STN platforms* (see Section 7.3). The characteristic that we find most important is *openness*: David's social TV, for instance, can use ThingsNet to store relations to social TVs on other STN Boxes (see Section 9.2), or it can send direct messages to social TVs on other STN Boxes.

8.4 Summary

In this chapter, we presented the current validations of our approach to create a world-wide *socio-technical graph (STG)* that is sustained by heterogeneous platforms and that machines can interpret and manipulate in a reliable fashion. The world-wide STG represents the backbone of our vision for a *Social Web of Things (SWoT)*. Our approach enabled the successful integration of several well-known social platforms in the deployed STG, namely *Facebook*, *SoundCloud*, and *Twitter*, and of a WoT platform, that is *Dweet.io*. By integrating all these platforms into the SWoT, we have demonstrated that our approach copes well with *platform heterogeneity*.

In Section 8.1, we discussed the integration of the above-mentioned platforms into the SWoT. In Section 8.2, we presented ThingsNet, our implementation of a *Level 5 STN platform* (see Section 7.3). In Section 8.3, we reported on our experience with deploying an STG for the scenario introduced in Section 5.1.1 ("The Social TV") across the platforms discussed in this chapter, and we presented a Web application (a.k.a. the *STN browser*) that is able to interpret *STN description documents* in order to interface with heterogeneous STN platforms. The *STN browser* validates our approach of creating a *hypermedia-driven environment* for the SWoT: starting with the URL of a *SWoT profile* (see Section 7.1.2) as an entry point into the SWoT, we are able to transcend platform boundaries to navigate and manipulate the world-wide STG in a uniform fashion.

Chapter 9

Bringing Rational Agents to the Social Web of Things

Contents

In the previous chapter, we validated our approach to create a *hypermedia-driven environment* sustained by *heterogeneous platforms*. Per our vision presented in Chapter 5, two important characteristics of this environment are that it supports *discoverability* and *flexible interaction* in the SWoT. In this chapter, we demonstrate these characteristics by implementing the scenarios presented in Section 5.1. A secondary objective of this chapter is to demonstrate that the abstraction layers introduced by our SWoT architecture (see Section 5.3) enable developers and users to effectively cope with the envisioned complexity of the overall ecosystem.

This chapter is structured as follows. In Section 9.1, we introduce a multi-agent middleware for the SWoT that aims at facilitating the development of *social things* (see Definition 6.1.7) as *rational agents*, that is to say software agents that can autonomously make decisions and act upon them. We use this middleware throughout the rest of this chapter to implement the scenarios presented in Section 5.1. In Section 9.2, we implement the "Social TV" scenario (see Section 5.1.1) to showcase how STNs can enhance *discoverability* in the IoT. In Section 9.3, we implement the "The Wake-up Call" scenario (see Section 5.1.2) to showcase how STNs can enhance *flexible interaction* in the IoT. In Section 9.4, we implement the "The Laundry Room" scenario (see Section 5.1.3) to showcase how STNs can serve as a uniform mechanisms for remote interaction with *heterogeneous* things. In Section 9.5, we implement the "A Welcoming Home" scenario (see Section 5.1.4) to showcase the use of regulation mechanisms for manipulating relations in the SWoT and coordinating the behavior of social things towards achieving common goals.

9.1 A Multi-agent Middleware for the Social Web of Things

In this section, we introduce the elements required to understand our scenario implementations in the rest of this chapter, and we present the details of how the agents we implement are able to participate into the SWoT such that throughout the rest of this chapter we can focus on programming the agent behavior instead.

We developed our multi-agent middleware for the SWoT using the JaCaMo platform [Boissier 2013]. We have chosen to use JaCaMo because it integrates three multi-agent platforms that provide all the elements needed to implement the applications presented in this chapter, that is to say:[1] Jason [Bordini 2007] for programming *BDI agents*, CArtAgO [Ricci 2009] for programming *multi-agent environments* using the Agents & Artifacts meta-model [Omicini 2008], and Moise [Hubner 2007] for programming *multi-agent organisations*.

Our middleware provides developers with implementations for the *social artifacts* defined by our *digital STN model* (see Section 6.3.3). The middleware handles the heavy lifting of interfacing with heterogeneous STN platforms and allows developers to focus on the design and implementation of SWoT applications in terms of the higher-level abstractions of *agents*, *artifacts* and *organisations*. Developers can also extend the middleware with new types of artifacts.

An overview of a typical JaCaMo application that can be developed using this middleware is depicted in Figure 9.1. Developers program *social things* as rational agents situated in *working environments* that span across the physical-digital space. A *working environment* is composed of a dynamic set of *artifacts* organized in *workspaces* (see [Ricci 2007b] for more details). As illustrated in Figure 9.1, agents perceive changes in their environment by observing artifacts, and act on their en-

[1]A more detailed discussion on the various modeling dimensions for multi-agent systems is available in Section 4.1.2

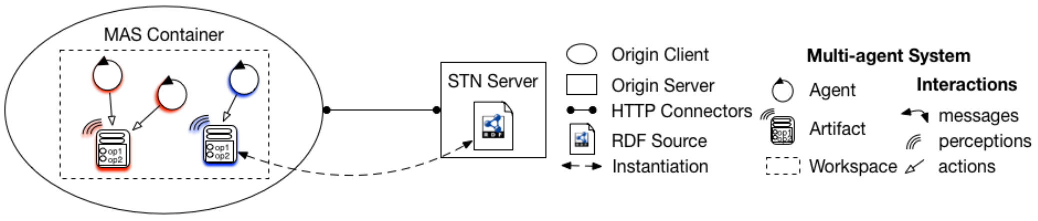


Figure 9.1: A multi-agent SWoT application. The application runs on an STN client and is composed of three rational agents and two artifacts. Two of the agents observe and act on a non-persistent artifact, while the third observes and acts on an artifact persisted on an STN server.

vironment by performing operations on artifacts. *Social artifacts* are persisted on STN platforms. Other artifacts used in our scenario implementations in this chapter are non-persistent and exist only at run-time.

9.1.1 Programming social things as BDI agents

In the applications presented in this chapter, we program social things as *belief-desire-intention (BDI) agents* [Bratman 1988]. *Jason* provides a customizable BDI agent architecture and a language for programming the agent's behavior[2]. In what follows, we discuss only elements of the Jason agent programming language that are necessary to understand the rest of this chapter. Thorough discussions on programming BDI agents using Jason are available in [Bordini 2006, Bordini 2007].

The basic language constructs provided by Jason to program an agent's behavior are:

- *beliefs*, which represent information an agent holds about the world. Beliefs are not necessarily true, they may be out of date or inaccurate.
- *goals*, which represent states of affairs an agent wishes to bring to the world.
- *plans*, which represent courses of actions an agent expects would achieve specific goals.

The main benefit of programming *social things* using Jason, or other platforms for programming BDI agents, is that developers are generally concerned only with programming the *behavior of social things* at a very high level of abstraction in terms of beliefs, goals, and plans. Furthermore, given the intuitive nature of these concepts, we suggest that *easy-to-use development tools*[3] would enable tech savvy users to

[2]The language provided by Jason extends AgentSpeak(L) [Rao 1996], an abstract programming language for BDI agents. A detailed discussion on the extensions that Jason brings to AgentSpeak(L) is available in [Bordini 2006].

[3]Such as a Web-based integrated development environment (IDE) seamlessly integrated in *David's STN Box* (see scenarios in Section 5.1).

program simple behaviors for social things without requiring advanced expertise in multi-agent systems.

Henceforth, we refer to agents implemented using the Jason platform as *Jason agents*. Jason agents are goal-driven and run continuously in *reasoning cycles*. During a reasoning cycle, a Jason agent perceives changes in its environment and reacts to events by executing plans intended to achieve goals. Goals are central to agent-oriented programming.

A Jason agent typically starts with an initial set of goals to be achieved that represent its design purpose, a belief base and a library of plans. *Beliefs*, *goals* and *plans* can evolve throughout an agent's lifetime. *Beliefs* can be collected from the environment via sensors, they can be communicated by other agents, or they can be generated by the agent itself (i.e., mental notes). *Beliefs* can also be inferred from an agent's existing belief base via rules. The library of *plans* can evolve, for instance, by learning new plans from other agents. *Goals* can be generated by plans in the pursuit of a given *goal* (i.e., a generated *subgoal*), or they can be delegated by other agents.

Jason agents are situated and "live" in *multi-agent environments*. We discuss the development of multi-agent environments for the SWoT in what follows.

9.1.2 Multi-agent environments for the SWoT

The major feature that our extension brings to the standard JaCaMo distribution is the ability to create, retrieve and manipulate artifacts persisted on *heterogeneous STN platforms*. We followed two design considerations for the implementation of this feature:

R1 working with *social artifacts* should add minimal overhead for developers;

R2 any extensions for the SWoT should not diverge from the standard JaCaMo distribution.

The first requirement implies that, for instance, the states of *social artifacts* should synchronize seamlessly with STN platforms. For another example, when a Jason agent receives a message, the transmitted information is automatically reflected in its belief base (see [Bordini 2007] for details). Receiving messages via an STN should be reflected in an agent's belief base much in the same way.

The second requirement implies that our SWoT middleware should not rely on custom builds of the JaCaMo platform such that it can benefit from future platform releases. For instance, per the first requirement, the most seamless mechanism to retrieve *social artifacts* from the Web would be to overload the look-up operation provided by CArtAgO workspaces, however this change would impact the CArtAgO *workspace artifact* and thus diverge from the standard JaCaMo distribution.

In the following, we discuss the main features of the SWoT middleware that we use to implement our application scenarios. We discuss working with social artifacts in Section 9.1.2.1 and communication via STN platforms in Section 9.1.2.2.

9.1.2.1 Working with social artifacts

In a JaCaMo application, Jason agents are situated in *working environments* that are modeled and programmed by means of CArtAgO artifacts (cf. Figure 9.1).

Agents can *act* on CArtAgO artifacts by invoking *operations* [Ricci 2009]. Henceforth, to avoid confusion, we generally refer to the operations that are part of a CArtAgO artifact's usage interface as *artifact operations*, and to the *operations* defined by our *digital STN model* (see Definition 6.1.11) as *STN operations*. If the meaning of the term is obvious in the context in which it is used, we use simply *operation*.

Our middleware currently provides four types of artifacts:

- `UserAccountArtifact`, which is a *social artifact* that wraps *user accounts* (see Definition 6.1.26);
- `MessageArtifact`, which is a *social artifact* that wraps *digital messages* (see Definition 6.1.28);
- `SWoTProfileReaderArtifact`, which is a helper CArtAgO artifact that agents can use to retrieve and inspect *SWoT profiles* (see Section 7.1.2), for instance to extract the *user accounts* held by an *agent* (see Definition 6.1.6);
- `STNPlatformDescriptionReaderArtifact`, which is a helper CArtAgO artifact that agents can use to retrieve and inspect *STN description documents* (see Section 7.1.3), for instance to extract the required parameters of an operation.

Performing STN operations

The main benefit that our middleware brings to developers is that it hides all the interaction with STN platforms. Developers are concerned only with the higher-level, uniform interface exposed by the *social artifacts* provided by our middleware. When an *artifact operation* is performed that involves one or more *STN operations*, the middleware handles composing and issuing the associated HTTP requests, and interpreting the HTTP responses, which includes the data integration step.

Creating social artifacts

A Jason agent can create a CArtAgO artifact in its current workspace by invoking the `makeArtifact` operation provided by a *CArtAgO workspace artifact*, which is part of the CArtAgO infrastructure. When the artifact is created, the artifact's *initialization method* is invoked with a list of parameters provided via the `makeArtifact` operation.

Per R1, we overload the initialization methods of our *social artifacts* for both creating new *social artifacts* and for retrieving existing ones from the SWoT. This approach enables developers to work with *social artifacts* in a similar manner as they would with regular CArtAgO artifacts.

When creating a *social artifact*, the artifact's initialization method requires:

1. the path to a configuration file that provides the performing agent's authentication data, which is necessary for signing HTTP requests (if the case); this practice discourages hard-coding credentials in the agent logic;

2. the URI of the STN platform on which the artifact is to be created; this URI should dereference to the platform's *STN description document* (see Section 7.1.3);

3. the parameters for the *CreateUserAccount* operation (see Section 6.3.3), as implemented by the targeted platform.

For illustrative purposes, Listing 9.1 shows how an agent can create a user account on ThingsNet by invoking the **`makeArtifact`** CArtAgO primitive. The parameters transmitted to the social artifact's initialization method begin on line 4 in the form of a *list* [Bordini 2007] that contains the path to the performing agent's configuration file, the URI of the platform, a list with the names of the parameters, and a list with the values of the parameters. Any term that starts with an uppercase letter is a variable.

```
 1 // Creating a user account on a ThingsNet node.
 2 makeArtifact("acc1",
 3     "webartifacts.UserAccountArtifact",
 4     [AuthFilePath, "http://localhost:9000/assets/stnspecs/thingsnet.ttl
         #platform",
 5         ["stn-ops:SocialThingClass",
 6             "stn-ops:SocialThingOwner",
 7             "stn-ops:DisplayedName"
 8         ],
 9         ["http://example.org/#socialTV",
10             "http://localhost:9000/users/e75...8f3#David",
11             "Test TV"
12         ]
13     ],
14     Acc
15 );
```

Listing 9.1: Using the **`makeArtifact`** CArtAgO primitive to create a user account on ThingsNet.

Retrieving social artifacts from the SWoT

When retrieving an existing social artifact from the SWoT, the initialization of a *social artifact* requires:

1. a path to the configuration file of the performing agent (see above);

2. an artifact URI *or* a platform URI and a platform-specific identifier.

For illustrative purposes, Listing 9.2 shows an invocation of the `makeArtifact` CArtAgO primitive to retrieve a *user account* from ThingsNet via the account's URI.

```
// Retrieving a user account from a ThingsNet node.
makeArtifact("acc1",
    "webartifacts.UserAccountArtifact",
    [AuthFilePath, "http://localhost:9000/users/e75...8f3#David"],
    Acc
);
```

Listing 9.2: Using the `makeArtifact` CArtAgO primitive to retrieve a user account from ThingsNet.

Crawling the SWoT

Social artifacts provide *artifact operations* that can result in retrieving additional *social artifacts* that are directly added to the current workspace.

For illustrative purposes, Listing 9.3 shows how an agent can retrieve a Twitter account and then retrieve the account's outgoing social relations, which is an implementation of the *GetOutgoingRelations* operation type (see Section 6.3.2). The retrieved Twitter accounts are automatically wrapped in *social artifacts* and added to the current workspace. A list of CArtAgO artifact identifiers is returned as an output parameter. This approach, as opposed to returning only references to Twitter accounts for instance, enables agents to manipulate the returned artifacts in a direct and simple manner, and thus without additional overhead for developers (per R1).

```
// Retrieving a user account from Twitter.
makeArtifact("tw1",
    "webartifacts.UserAccountArtifact",
    [AuthFilePath,
        "http://localhost:9000/assets/stnspecs/twitter.ttl#platform",
        "swotwm"
    ],
    Tw
);

// Retrieving the outgoing connections of a Twitter account.
getOutgoingRelations(Conns)[artifact_id(Tw)];
```

Listing 9.3: This listing shows the retrieval of the Twitter accounts followed by a given user.

9.1.2.2 Communication via STNs

As noted in Section 5.2.1.2, existing social platforms generally function as central brokers for messages exchanged between their users. *Social things* can communicate via STNs by posting public messages, which are then routed by the platform to subscribers, or by sending direct messages to other *social things*. In the applications presented in this chapter, we use Twitter to disseminate information among *social things*. To this purpose, the *user account* CArtAgO artifact provided by our SWoT middleware implements an artifact operation for posting messages to STN platforms, and an artifact operation that activates/deactivates the receiving of new messages posted by publishers the user account is subscribed to.

For illustrative purposes, Listing 9.4 shows a *Jason plan* [Bordini 2007] for receiving and replying to a new message. A Jason plan is composed of a *triggering event*, a *context* in which the plan can be applied, and the plan's *body*. In Listing 9.4, the triggering event is that the agent received a new message on a given *platform* and from a given *user account*. The agent prints the message to the standard output, together with the name of the sender, and post an update on the STN.

```
+newFeedMessage(PaltformUri, SenderAccount, Message) : true <-
    getName(PresumedSender)[artifact_id(SenderAccount)];
    .print("New message: ", platformUri, " ", PresumedSender, " ",
        Message);
    postMessage("A Jason agent with a message!")[artifact_id(Tw)].
```

Listing 9.4: A Jason plan for replying to a message received via an STN.

Our middleware provides an *internal action* [Bordini 2007] that agents can use to interpret messages received via STNs and update their belief base accordingly.

Having introduced all the basic elements used by our middleware, in the following sections we report on our implementations of the scenarios presented in Section 5.1.

9.2 Use Case: Crawling the Social Web of Things

We have implemented the "Social TV" scenario (see Section 5.1.1) using a Jason agent to implement David's social TV and the world-wide STG deployed in the previous chapter (see Section 8.3). The purpose of this application scenario is to showcase how the SWoT enables *discoverability* by searching the world-wide STG in an informed manner.

9.2.1 Deployment Scenario

The world-wide STG used in this application scenario is deployed across Facebook, Twitter, SoundCloud, Dweet.io, and multiple instances of ThingsNet. In addition to the setup already presented in Section 8.3.1, for this application scenario we inject movie ratings on predefined Dweet.io user accounts (see Section 8.1.4) to simulate the data published by the social TVs of David's friends.

The Jason agent is given the URIs of David and his STN Box, which we assume are obtained from David during the installation of the social TV, and has to achieve the following tasks:

1. Create a *user account* on the STN Box, i.e. ThingsNet.

2. Crawl the SWoT, using David's URI as an entry point, to discover and create *social relations* to other social TVs owned by David's friends.

3. After the crawling phase is completed, search all *user accounts* that are *held by* known social TVs and are *hosted by platforms* that *support* the *operation type* denoted by `stn-ops:GetUserAccountFeed`.

4. For each *user account* discovered at the previous step, retrieve the latest *messages* posted to its feed.

Dereferencing David's URI retrieves a representation of his *SWoT profile* presented in the previous chapter (see Listing 8.12), which is also his WebID profile document [Sambra 2015a]. Dereferencing the URI of the STN Box retrieves a representation of ThingsNet's *STN description document* (see Appendix A).

The agent is also given three more preconfigured parameters: its *social thing class*, which is `http://example.com#SocialTV`, and a *name* and *description* to be displayed within STNs. The agent has a file with all the authentication data it needs to sign HTTP requests in order to access the social platforms (see Section 8.1).

9.2.2 Agent logic

The agent uses the `STNPlatformDescriptionReaderArtifact` artifact (see Section 9.1.2) to extract from ThingsNet's *STN description document* the parameters required to perform the *CreateUserAccount* operation type, that is (see Section 8.2.2): its social thing class, the WebID of its owner, and a name to be displayed within the STN. It then collects the required parameters from its initial belief base and creates a `UserAccountArtifact` social artifact using these parameters and the ThingsNet's URI (see Section 9.1.2).

After registering to ThingsNet, the agent implements Algorithm 1 to crawl the SWoT and create *social relations* to other social TVs. To construct David's distributed social graph, the agent uses the `SWoTProfileReaderArtifact` artifact (see Section 9.1.2) to extract from David's *SWoT profile* the metadata required to retrieve representations of his user accounts on the various platforms he uses (cf. Listing 8.12), and then retrieves the outgoing social relations of each *user account.* If David's SWoT profile also includes social relations, these are added to his distributed social graph.

For each of David's friends or user accounts that represent one in his distributed social graph, the agent retrieves the entity's description to extract all *things* that are *owned by* David's friend. If the entity is a *user account* and its description contains the friend's *SWoT profile*, which is the case in our deployment scenario via the `website` field (see discussion in Section 8.1), the agent constructs the friend's distributed social graph as it did for David.

Algorithm 1 The crawling algorithm implemented by our Jason agent to discover and connect to social TVs owned by David's friends.

```
for all agents/user accounts in David's distributed social graph do
  if entity description contains a SWoT profile then
    retrieve the SWoT profile
    construct the entity's distributed social graph
  for all things owned by this entity do
    if thing is a social TV then
      create a social relation to thing
```

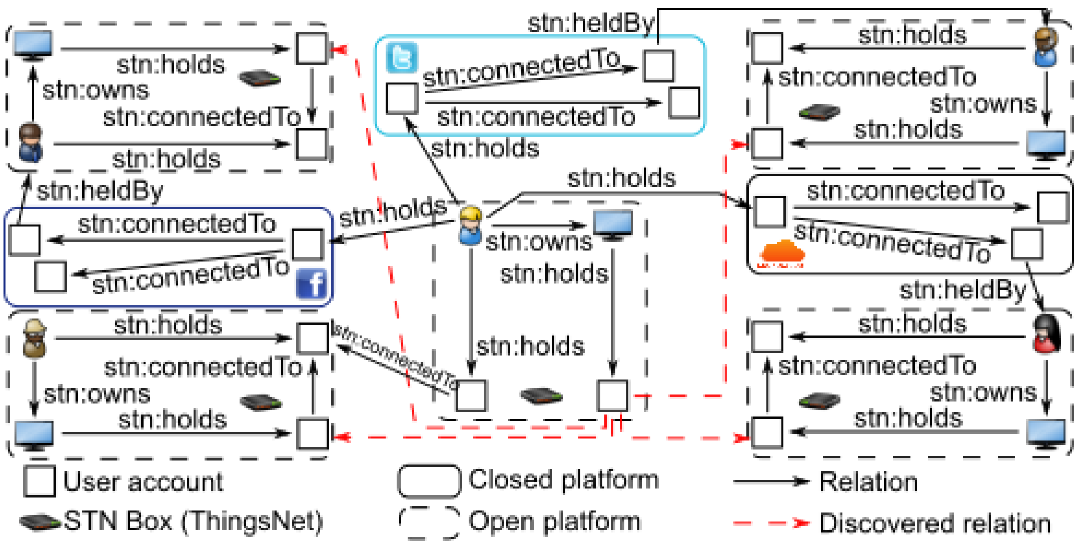


Figure 9.2: Results of crawling the deployed STG. The social TV has discovered and created *social relations* to four other TVs owned by David's friends.

For all discovered *things*, if the thing is a social TV, that is the thing's class is also denoted by `http://example.com#SocialTV`, the Jason agent creates a *social relation* to the *thing* using the ThingsNet *user account* created previously. This completes the crawling phase. The results are illustrated in Figure 9.2.

In a similar manner, the agent searches its own graph to discover any user accounts *held by* known social TVs. For each discovered *user account*, the agent then performs the `getUserAccountFeed` *artifact operation*, which is implemented by the `UserAccountArtifact` social artifact (see Section 9.1.2). If the *user account*'s hosting platform does not support the operation type denoted by the `stn-ops:GetUserAccountFeed` URI, the action fails. It is important to emphasize that, in doing so, the *Jason agent is agnostic to the underlying hosting platforms.*

9.2.3 Lessons learned

An important observation based on this scenario implementation is that our approach adds value to the development of IoT applications by integrating platforms with complementary functionality. In this particular scenario, the social TVs publish data via Dweet.io, a WoT platform (see Section 8.1.4). Dweet.io is a necessary addition to this setup: social TVs could also use Twitter to obtain the same functionality, however Dweet.io facilitates publishing structured data and keeps the data feeds out of the way of human users. Furthermore, Twitter restricts messages to 140 characters (see Section 8.1.3). On the other hand, without integrating Dweet.io into the SWoT, the social TVs' data feeds *and* the Dweet.io platform itself would *not be discoverable*. Both the Dweet.io data feed URIs and the platform-specific knowledge required to access them would have to be hard-coded into *social things*.

Second, it is worth to notice that, given a proper SWoT middleware, developers can easily implement complex functionality, such as crawling heterogeneous STNs. Our middleware handles the heavy lifting of interfacing with heterogeneous platforms and developers can focus on implementing the agent's logic, which treats the heterogeneous STNs in a uniform fashion. Furthermore, we expect that crawling would be a commonly used functionality in the SWoT, and therefore additional libraries, possibly implementing various crawling strategies, could further streamline the implementation of SWoT applications. This aspect also opens up an interesting research topic to be investigated, that is strategies for informed searches in the SWoT (see Section 10.3).

A challenge for the development and testing of this scenario implementation was posed by API rate limits. For instance, the Twitter Public API v1.1 permits at most 15 authorized requests in a 15 minutes interval to most endpoints.[4] This rate limit is high enough to crawl the tiny Twitter graph deployed in our scenario, however it sniffles the crawling of large graphs or the testing of such applications. We expect that access to resources would be a constant challenge in the SWoT, which raises the more general research question of defining strategies that *social things* could use to share resources or compete for shared resources (see Section 10.3). On the one hand, one could imagine that social things owned by the same user could, for instance, share cached data or access to various APIs in order to mitigate rate limiting. On the other hand, an STN Box could limit the outgoing or incoming traffic, therefore causing social things to compete for access to outside STNs.

9.3 Use Case: Flexible Interaction among Social Things

We have implemented the "Wake-up Call" scenario (see Section 5.1.2) using multiple Jason agents to implement David's social things, and Twitter to implement his STN Box. The purpose of this application scenario is to showcase:

- *flexible interaction* by means of *agent interaction protocols* (see Section 4.2.2.3);

[4]https://dev.twitter.com/rest/public/rate-limits, Accessed: 28.11.2015.

- the use of the STN paradigm and STN platforms as *information dissemination mechanisms* for *social things* that are intuitive for both developers and non-technical users;
- the added value of using agent programming languages, such as Jason [Bordini 2007], as *domain-specific languages for programming social things*.

9.3.1 Deployment scenario

In this scenario, David's social things can interact in a flexible manner to wake him up if he is sleeping and there is an upcoming meeting. To this purpose, the social things produce and/or consume contextual information about David's bedroom, which we represent in terms of *beliefs*. For instance, whenever going to bed or waking up, David can monitor his sleep by pressing a button on his wristband to switch between the day and night operating modes. David's actions result in beliefs that the wristband holds about David being asleep or not. Beliefs can carry annotations, such as the *source* that originated the belief (e.g., the wristband) and the *degree of certainty* the source has in that belief, which in our scenario implementation is represented as a decimal number in the `[0,1]` interval. The contextual information that can be produced or consumed by the social things in this scenario is shown in Table 9.1.

Table 9.1: Contextual information produced or consumed by the social things in our scenario implementation.

Social Thing	**Produces**	**Consumes**
Calendar	-	asleep(david)
Wristband	asleep(david)[certainty=0.8]	-
Curtains	curtains_state(open), curtains_state(closed)	asleep(david), outside_light_level(Level)
Lights	lights(on), lights(off)	asleep(david), curtains_state(State), outside_light_level(Level)

The social things are all connected to one another via Twitter and can thus broadcast *messages* to other things. The payload of a message published via the STN includes:

- a *performative*, which we choose among the ones defined by Jason [Bordini 2007];
- a *propositional content* that denotes the object of the action.

In particular, in our implementation we use two performatives [Bordini 2007]:

- `tell`, which denotes that the sender of a message believes the transmitted *propositional content* is true and intends for the receivers to believe so as well;

- `untell`, which denotes that the sender of a message *does not* believe the transmitted *propositional content* is true and intends for the receivers not to believe as well.

For instance, a social thing can launch a *call for proposals* to wake up David by posting the following message[5]: `tell cfp(CNPId, achieve(not asleep(david)))`, where `CNPId` is an identifier for the launched interaction. The *propositional content* is a Jason term [Bordini 2007] and we assume that social things rely on domain-specific ontologies or other conventions to interpret such messages.

It is worth to note that we model the social things in this scenario implementation after existing products, such as the Jawbone UP24 wristband[6], the Luna smart mattress cover[7], or the Luxone smart curtains[8] (cf. Table 9.1).

9.3.2 Agent logic

David's social things reason under the open-world assumption, that is to say whatever a social thing does not know to be true or false is simply unknown. We discuss the logic implemented by each social thing in what follows.

The social calendar

The social calendar plays a central role in this scenario. If the calendar believes that David is asleep and thus in danger of missing a scheduled meeting, it interacts with the other social things in the home STN with the goal of waking him up. The calendar is blindly committed to this goal, meaning that it will continue to attempt to wake up David until it believes that David woke up. In order to determine if David is asleep or not, the calendar relies on information published via the STN Box by other social things, such as the wristband and mattress cover (cf. Table 9.1).

The calendar's initial set of beliefs is shown in Listing 9.5. The calendar believes that it is owned by David and that there is a meeting scheduled on December 1, 2015, at 10:00. The calendar also believes that David prefers the most to be awakened up by natural light, and the least by sound alarms.

```
owned_by(david).
meeting("m1", 10, 00, 2015, 12, 01).

alarm_rank(vibrations, 1).
alarm_rank(natural_light, 0).
alarm_rank(artificial_light, 2).
alarm_rank(sound, 3).
```

Listing 9.5: The calendar's initial set of beliefs.

[5]We discussed agent interaction protocols in Section 4.2.2.3.
[6]https://jawbone.com/up/, Accessed: 26.09.2015.
[7]http://lunasleep.com/, Accessed: 26.09.2015.
[8]http://www.loxone.com/enen/smart-home/everything-managed/curtains-and-blinds/curtains.html, Accessed: 26.09.2015.

The Jason plans for monitoring David's meetings are shown in Listing 9.6. The calendar starts a new interaction to wake up its owner if it believes there is an upcoming meeting and its owner is asleep. The rule `upcoming_meeting(M)` infers from the calendar's belief base if there is a meeting `M` scheduled in less than one hour. The calendar infers that its owner is asleep if it holds such a belief with a degree of certainty above `0.5`.

```
+!monitor_scheduled_meetings : upcoming_meeting(M) & owned_by(Owner) &
    asleep(Owner)[certainty(C)] & C > 0.5 <-
    .print("Upcoming meeting ", M, " and ", Owner, " is asleep.");
    jia.genUUID(CNPId); // generate a UUID for this CNP
    // start and manage the interaction
    !startCNP(CNPId, achieve(not asleep(Owner))).

// keep monitoring
+!monitor_scheduled_meetings : true <- !monitor_scheduled_meetings.
```

Listing 9.6: The calendar's plans for monitoring his owner's meetings.

If David is asleep and there is a meeting scheduled in less than one hour, the calendar initiates an interaction implementing the Contract-Net Protocol (see Section 4.2.2.3 for details) by posting via Twitter the *call for proposals (CFP)* presented in the previous section in order to wake up David. Social things that can perform this action reply with proposals containing their wake up methods.

The calendar waits for a predefined amount of time to receive proposals and selects one in accordance with its David's preferences. If no proposals are received or the wake up attempt fails (i.e., there are no updates from other social things that *David is not asleep* anymore), the calendar reinitiates the interaction.

The social wristband

When David changes the function mode of his wristband from day to night to record a new sleeping session, the wristband posts a message via the home STN with the payload `tell asleep(david)[certainty(0.8)]`. Similarly, when David changes the function mode from night to day, the wristband posts a message with the payload `untell asleep(david)`.

We simulate pushing the mode button via beliefs originated from a percept. These beliefs are added to/removed from the wristband's belief base, which activates the corresponding plans shown in Listing 9.7. For instance, if the mode is changed from day to night, the wristband takes a mental note that David is asleep with a certainty of 0.8, and posts the confirmation via the STN Box.

```
+function_mode(night)[source(percept)] : owned_by(Owner) <-
    +asleep(Owner)[certainty(0.8)];
    .concat("tell asleep(", Owner, ")[certainty(0.8)]", Message);
    postMessage(Message).

-function_mode(night)[source(percept)] : owned_by(Owner) <-
    -asleep(Owner);
    .concat("untell asleep(", Owner, ")[certainty(0.8)]", Message);
    postMessage(Message).
```

Listing 9.7: The wristband's plans for function mode changes.

If the wristband receives a CFP to wake up its owner, and it believes the owner is asleep, it posts its proposal to wake up David via its vibration alarm. The Jason plan implemented for this purpose is shown in Listing 9.8. It is worth to note that this plan triggers only if the the wristband believes that indeed David is asleep. This implies that the wristband is in night mode and, if so, we assume it is likely that it is also on David's wrist.

```
+cfp(CNPId, achieve(not asleep(Owner))) : owned_by(Owner)
    & asleep(Owner) <-
    // add the proposal to the belief base (mental note)
    +proposal(CNPId,achieve(not asleep(Owner)),alarm_type(vibrations));
    // construct and post message
    .concat("tell propose(", CNPId, ", alarm_type(vibrations))",
        Message);
    postMessage(Message).
```

Listing 9.8: The wristband's plan for posting proposals.

If the proposal is accepted, the wristband sets off its vibration alarm to wake up David, which we implement by means of two Jason plans shown in Listing 9.9. The `wakeup_attempt` rule is used to simulate the wake-up attempt with a 50% chance of success.

```
+!set_off_vibrations_alarm(CNPId) : wakeup_attempt(true) <-
    -+function_mode(day)[source(percept)]; // simulate mode change
    .concat("tell inform_done(", CNPId, ")", Message);
    postMessage(Message).

+!set_off_vibrations_alarm(CNPId) : wakeup_attempt(false) <-
    .concat("tell failure(", CNPId, ")", Message);
    postMessage(Message).
```

Listing 9.9: The wristband's plans for waking up David.

It is worth to note that the wristband's belief that David woke up may, in fact, be false. For instance, David can cancel the alarm, switch the function mode, and go back to sleep. Luckily for David, however, he also has a social mattress cover.

The social mattress cover

The social mattress cover functions similarly to the wristband, with the difference that it uses sensors to detect if David is in bed or he got out of bed, and therefore has a higher degree of certainty in its beliefs (cf. Table 9.1). The mattress cover can only provide information to other social things, it cannot wake up David in any way.

The social curtains

Listing 9.10 shows the plan implemented by the social curtains in order to respond to calls for proposals that it has learned about. The social curtains propose to wake up their owner only if (i) they hold a belief he is asleep that originated from the mattress cover, which we assume implies that David is sleeping in his bedroom, (ii) the curtains are not already open, and (iii) outside light level is above 100 lux (S.I.), which is the equivalent of a very dark day[9]. We assume that the curtains can get readings of outside light levels on-demand from a light sensor.

```
daytime(V) :- outside_light_level(L) & .eval(V, L >= 100).

+cfp(CNPId, achieve(not asleep(Owner))) : owned_by(Owner)
    & asleep(Owner)[source(mattress_cover)]
    & curtains_state(closed) & daytime(true) <-
    // mental note to remember my proposal
    +proposal(CNPId, achieve(not asleep(Owner)), alarm_type(
        natural_light));
    // post proposal
    .concat("tell propose(", CNPId, ", alarm_type(natural_light))",
        Message);
    postMessage(Message).
```

Listing 9.10: The social curtains' plan for posting proposals.

The social lights

The context in which the social lights in David's bedroom offer to wake up David is complementary to the one addressed by the social curtains, that is if the outside light level is below 100 lux (S.I.), or if it is above and the curtains are closed (cf. Listing 9.11). The lights rely on information posted by the curtains and readings of outside light levels.

```
+cfp(CNPId, achieve(not asleep(Owner))) : owned_by(Owner)
    & asleep(Owner)[source(mattress_cover)]
    & (daytime(false) | (curtains_state(closed) & daytime(true))) <-
    // mental note to remember my proposal
    +proposal(CNPId, achieve(not asleep(Owner)), alarm_type(
        artificial_light));
    // post proposal
    .concat("tell propose(", CNPId, ", alarm_type(artificial_light))",
        Message);
    postMessage(Message).
```

Listing 9.11: The social lights' plan for posting proposals.

The social smartphone

The social smartphone is a mobile device and relies on sound to wake up its owner, therefore it can respond with a proposal in any context. The logic implemented for

[9]http://www.engineeringtoolbox.com/light-level-rooms-d_708.html, Accessed: 27.09.2015.

the smartphone is similar to the one presented for the other social things.

9.3.3 Lessons learned

This scenario implementation illustrates the use of agent interaction protocols to enable *flexible interaction* among things. The main benefit of this approach, as opposed to a static IoT mashup, is that things are decoupled and can thus be deployed and can evolve independently from one another, which provides for a flexible way of constructing smart environments. One could also imagine a process-driven composition of services that integrates agent interaction protocols to discover or auction for service providers.

Developers can easily program or extend social things with new functionality, given appropriate paradigms and frameworks. The social things in this scenario implementation rely on simple plans for reacting to changes in their environment, such as calls for proposals launched by the social calendar. Furthermore, given that social things are programmed at a very abstract and intuitive level, that is in terms of beliefs, goals, and plans, we hypothesize that, given the right tools, tech savvy users without expertise in multi-agent systems could, in fact, program smart environments. For instance, David's STN Box could expose a Web front end that allows David to add new plans to his social things. Simple plans could be created via an advanced graphical user interface.

It is worth to note that STNs can be used as mechanisms for logging interactions in smart environments. Human users can then inspect such interactions in an already intuitive and familiar fashion. This aspect opens up an interesting research topic, that is explaining actions in SWoT environments (see Section 10.3).

In our implementation, all social things follow one another. As the number of social things increases, it becomes increasingly important for social things to manage their relations such that they optimize the flow of information in the STN. For instance, there is little reason for the wristband to subscribe to updates from the social curtains in our scenario. Social things could choose their relations, for instance, based on the contextual information produced or consumed by other social things, or based on their interaction history with other social things. This aspect opens up an interesting research topic, that is managing relations in the SWoT (see Section 10.3).

9.4 Use Case: Remote Interaction with Social Things

We have implemented the "Laundry Room" scenario (see Section 5.1.3) using Jason agents to implement the washing machines and Twitter to enable interaction with users. The purpose of this application scenario is to showcase the use of digital STNs as uniform mechanisms for remote interaction with heterogeneous things: Andrei can post a tweet to discover available washing machines, receive replies and reserve a time slot for doing his laundry, as opposed to the trial-and-error approach of visiting each laundry room on each floor of his student house (see Section 5.1.3 for more details).

9.4.1 Deployment Scenario

The interaction pattern in this scenario is similar to one in the "Wake-up call" scenario in the previous section, with the difference that in this scenario a *person* initiates the interaction with a call for proposals for doing the laundry.

We run two instances of the same Jason agent to implement two washing machines. A person can interact with the agent by tweeting "laundry". The agents reply with their availability.

9.4.2 Agent logic

The plans implemented by the Jason agent to process and reply to tweets are shown in Listing 9.12. The agent ignores any messages that are not "laundry", and replies to the other by proposing a time.

```
+!process_timeline([A|Authors], ["laundry"|Statuses]) : true <-
    .print("Got laundry command from ", A);
    .time(H,M,_);
    .concat("@", A, " ", "I can do the laundry at ", H, ":", M + 10,
        Reply);
    postMessage(Reply);
    !process_timeline(Authors,Statuses).

// got a tweet that is not a laundry cfp
+!process_timeline([A|Authors], [S|Statuses]) : true <-
    !process_timeline(Authors,Statuses).

+!process_timeline([],[]).
```

Listing 9.12: The washing machines' plans for processing and replying to tweets.

9.4.3 Lessons learned

This scenario implementation illustrates the use of digital STNs as intuitive, uniform mechanisms for remote interaction with heterogeneous things. It also contrasts interactions in natural language to the previous agent communication language interactions. In an STN platform, one can imagine that both could be used, one to be displayed to people, and the other to be consumed by agents. We discuss this further in the future work section (see Section 10.3).

In this implementation, messages are grouped in a thread that users can easily inspect. This behavior, however, requires that agents hard-code a feature that is specific to the Twitter platform. To reply to a tweet, the *operation* can include a *parameter* with the platform-specific identifier of the tweet, but it must also include in the reply the *Twitter screen name* of the author of the tweet, otherwise the parameter is ignored.[10] Our model and STN ontology can handle the first condition, however the latter cannot be expressed at the moment. We leave it as future work to further investigate actions related to social media, such as creating threads of messages, promoting and disseminating content (see Section 10.3).

9.5 Use Case: Regulation in the Social Web of Things

We have implemented the "A Welcoming Home" scenario (see Section 5.1.4) using Jason agents to implement David's social things, MOISE+ [Hubner 2007] to design and implement a multi-agent organisation for David's home (see Section 5.3.4), and Twitter to implement David's STN Box. The purpose of this application scenario is to showcase the use of externally defined norms and regulation mechanisms for:

- manipulating relations among social things in the SWoT;
- coordinating the behavior of social things towards achieving common goals.

9.5.1 Organisational specification

In this application scenario, David leaves his office to head home, which is announced by his car via posting a message on his STN Box. David's social things coordinate to prepare a warm welcoming.

To implement this scenario, we use MOISE+ [Hubner 2007] to design a *normative organisation* for David's home environment. The complete organisational specification is available in Appendix B.

A MOISE+ organisation is designed on multiple dimensions, namely *structural*, *functional*, and *normative*. The *structural dimension* defines *roles* that agents can enact when entering the organisation, *links* among those roles, which can be *acquaintance*, *communication*, or *authority* links, and *groups* defined by means of *roles* and *role cardinalities*.

We define the structural dimension of our organisation as follows:

- *roles*: we define a root role, which is `social_thing`, and extend it to define roles for each type of social thing owned by David (e.g., `car`, `vacuum_cleaner`, `thermostat`);
- *links*: we define a single *communication link*, from `social_thing` to `social_thing`;
- *groups*: we define a single group specification, that is `home_group`, that contains exactly one role for each of type of social thing in this scenario.

The agents in our scenario implementation have predefined roles that they enact when joining the organisation. In the initial state of the application, there are no *social relations* between the agents.

[10] https://dev.twitter.com/rest/reference/post/statuses/update, Accessed: 03.12.2015.

The *functional dimension* of our organisation defines a single *scheme specification*, namely `welcome_home_scheme` (cf. Appendix B). The goal of preparing a *warm welcoming* is decomposed in multiple subgoals that can agents can achieve in *sequence* or in *parallel*. *Goals* are grouped in *missions*.

The *normative dimension* assigns *missions* to *roles* by means of *norms*, which can specify *obligations* or *permissions*. When an agent enacts a role, he must fulfill the obligations associated to that role. We discuss this further in the following.

9.5.2 Agent logic

We run our application scenario in three phases. In the *first phase*, a *home agent* instantiates a `home_group` group via an *organisational artifact* [Boissier 2013]. Once the group is available, each agent adopts its predefined *role*.

In the *second phase*, *social things* conform to the organisation's *structural specification* to "follow" one another on Twitter. The structural specification is available as an observable property of the *organisational artifact*.

In the *third phase*, the *car agent* posts a message that David is heading home. The *home agent* picks up the update, instantiates the `welcoming_home_scheme` scheme via an *organisational artifact*, and adds it to the previously created group. This action initiatives the scheme, and each agent has to fulfill its obligations.

In our implementation, agents are norm-compliant, that is to say they fulfill all their obligations. There are two types of obligations an agent can receive from a MOISE+ organisational scheme artifact [Hubner 2007]: to commit to a mission or to achieve a goal. For illustrative purposes, the plans implemented by each agent to deal with obligations are shown in Listing 9.13.

```
+obligation(Ag,Norm,committed(Ag,Mission,Scheme),Deadline)
     : .my_name(Ag) <-
     .print("I am obliged to commit to ",Mission);
     commitMission(Mission)[artifact_name(Scheme)].

+obligation(Ag,Norm,achieved(Scheme,Goal,Ag),Deadline)
     : .my_name(Ag) <-
     .print("I am obliged to achieve goal ",Goal);
     // Act on the environment to fulfill goal.
     // Notify that I have achieved my goal:
     goalAchieved(Goal)[artifact_name(Scheme)].
```

Listing 9.13: Plans implemented by the Jason agents in this scenario in order to fulfill their obligations.

9.5.3 Lessons learned

In this application scenario, we demonstrate the use of normative concepts, which are defined externally to the agent logic, for bootstrapping social things into an STN. By externalizing the definition of relations among social things, the structure of an STN can evolve independently of the agents, which is an important aspect in an open system in which no assumptions can be made about the logic of heterogeneous agents. Furthermore, this approach also simplifies the agent logic. For instance, a conforming agent would only need to implement a simple strategy of always fulfilling obligations to create or delete relations, if such obligations are received from authorized parties (e.g., its owner, hosting STN Box). If such a mechanism is not implemented, or the agent fails to follow its obligations, from the system's perspective it is easy to take measures against the agent, such as social exclusion or prohibiting access to the service. The approach we take in this scenario complements and may be used in conjunction with the one already discussed in Section 9.2 for the implementation of the "Social TV" scenario, in which rules and strategies for managing relations are embedded in the agents' logic.

The second motivation of this application scenario is to demonstrate the use of regulation mechanisms to coordinate the behavior of social things. Using MOISE+, we are able to define and decompose goals in terms of subgoals that may be achieved in sequence or parallel. This approach may seem similar to the process-driven approach typically used for creating physical mashups, however the autonomy of social things implies an important benefit that enables the development of more flexible IoT applications: autonomous agents can learn on-the-fly how to achieve goals, for instance by exchanging plans and functional schemes with other agents, or by constructing plans and functional schemes via automated planning, similar to the goal-driven composition of services discussed in Section 3.3.2. It is also worth to note that goals, goal decomposition, plans and actions are intuitive, high-level concepts that do not require a technical background, and therefore could also be used by non-technical users in order to program and customize their smart environments.

9.6 Summary

In this chapter, we addressed Research Question 4, that is *how to enhance discoverability and flexible interaction in the Web of Things*. To this purpose, we applied our approach to integrate heterogenous STN platforms into a world-wide *socio-technical graph (STG)* in order to create hypermedia-driven, multi-agent environments for the SWoT. We were then able to apply existing multi-agent technologies to bring *rational agents* to the SWoT. These agents are *autonomous*, and they can *autonomously* make decisions and act upon them. For instance, they can autonomously crawl the world-wide STG to *discover* other social things, while being agnostic to the underlying hosting platforms. They can autonomously manipulate the STG such that they can "rewire" their relations with other agents, therefore creating *flexible* networks of agents. They can autonomously use these relations to interact with other agents, therefore achieving *flexible interaction*. Agents, both human and non-human, are thus quite literally *weaving the Social Web of Things*.

In Section 9.1, we presented the multi-agent middleware for the SWoT that we use to develop the applications presented in this chapter. In Section 9.2, we reported on our implementation of the "Social TV" scenario (see Section 5.1.1) to showcase *discoverability* by means of informed searches on the world-wide STG (cf. Limitation 1 and Limitation 2). In Section 9.3, we reported on our implementation of the "Wake-up call" scenario (see Section 5.1.2) to showcase *flexible interaction* by means of agent interaction protocols (cf. Limitation 3). In Section 9.4, we reported on our implementation of the "Laundry Room" scenario (see Section 5.1.3) to showcase the use of STNs as uniform mechanisms for remote interaction with heterogenous things (cf. Limitation 4). In Section 9.5, we reported on our implementation of the "A welcoming home" scenario (see Section 5.1.4) to showcase the use of externally defined regulations and regulation mechanisms for enabling control over the evolution of the world-wide STG and achieving coordination among social things.

The scenario implementations presented in this chapter validate the foundational principles introduced in Chapter 5, which provide the undermining of our proposal, and demonstrate that are able to successfully address the four limitations that motivate our work (see Section 1.1).

Part IV

Conclusions and Perspectives

Chapter 10

Conclusions and Perspectives

Contents

The goal of our thesis is to enable *autonomous* and *flexible interaction* in a global Internet of Things (IoT) ecosystem. This requires that things are not confined to Web silos, they are discoverable, and they go beyond being manually wired in static IoT mashups. Furthermore, people should be able to manage, interact with and keep track of large numbers of heterogeneous, collaborative things in a uniform fashion.

We proposed to address these limitations by endowing things with *autonomy* and applying the *social network metaphor* to the IoT to create *flexible networks* of people and *autonomous things*. We envision an *open* and *self-governed* IoT ecosystem, which we call the *Social Web of Things (WoT)*, in which people and things are situated and interact in a *global environment* sustained by *heterogeneous platforms* that supports *discoverability* and *flexible interaction*. Autonomous things are *first-class citizens* of this ecosystem and, together with people, they are quite literally weaving the SWoT.

In this chapter, we summarize this dissertation, our contributions, and discuss perspectives on future research.

10.1 Summary

In this section, we present a brief summary of this dissertation. We discuss our contributions in further detail in the following section.

In Chapter 1, we introduced the motivation of our work, and we presented our thesis and the research questions that we address in this dissertation. This dissertation is structured in four parts.

In Part I, we analyzed the state-of-the-art to define in further detail the limitations that motivate our work and to identify related models and technologies that could help bring about the envisioned IoT ecosystem.

In Chapter 2, we presented the REST architectural style, which guided the development of the modern Web, we analyzed in further detail the problem of Web silos, and we discussed current developments in the Web research community. We now have the required standards and technology to enable application-layer interoperability in the IoT via the Web. However, if the IoT is to be a true global ecosystem, it is mandatory to find solutions that would mitigate the problem of Web silos. To the best of our knowledge, there are no proposals that would enable the structured integration of non-uniform, non-hypermedia APIs into a global, hypermedia-driven environment.

In Chapter 3, we discussed in further detail the limitations that motivate our work. Discoverability and searching the WoT, interacting with large numbers of heterogeneous devices, and enabling automatic composition of IoT mashups are open problems being investigated from various angles. Some of these approaches look at applying social concepts, and in particular social networks, to the IoT/WoT. However, in most cases, these approaches are limited to leveraging existing online social platforms, and in many cases they are focused on particular platforms. Other proposals go a step further and, in addition to social networks, also take into account endowing things with autonomous behavior. However, these proposals either do not provide solutions to endow things with such abilities or are limited to hard-coding a predefined set of rules into things to govern their behavior.

In Chapter 4, we looked at current results from multi-agent research that could be useful to endow things with autonomous behavior. We found that there are many existing paradigms and technologies that could be useful to this purpose, but also to endow things with sociability, that is the ability to autonomously interact with other entities, and to make them susceptible to regulation.

In Part II, we introduced our vision, models and solutions that could help bring about the envisioned IoT ecosystem.

In Chapter 5, we presented several application scenarios to help articulate our vision and define the properties and requirements for the SWoT. Based on the latter, we postulated four foundational principles that provide the underpinning of our approach, namely relying on a *RESTful architecture*, and enabling *social connectivity*, *autonomy*, and *regulation* in the SWoT. Guided by a set of design principles (i.e., generality, separation of concerns, interoperability), we reasoned up from these foundational principles to propose our layered architecture for the SWoT.

In Chapter 6, we defined in further detail the various dimensions of STNs. We formalized our discussion via a general mathematical model for STNs that can be used to model any network-like system of agents. We then extended this model with formal definitions for the digital dimension of an STN. The *digital STN model* provides an unambiguous foundation for the SWoT that things can use both to obtain reliably representations of STNs, and of the operations through which they can access and participate in the STN.

In Chapter 7, we applied our digital STN model to provide a semantic description framework for STNs. Developers can use this framework to integrate existing platforms into the SWoT as STNs, or in the development of new STN platforms. We presented solutions to achieve uniform interfaces for heterogeneous STNs and a progressive integration strategy for the development and integration of STNs into the SWoT. Hiding platform heterogeneity behind uniform interfaces is necessary in order to decouple things from their environments.

In Part III, we reported on the current validations of our work.

In Chapter 8, we reported on our experience with deploying a world-wide *socio-technical graph (STG)* across multiple existing social platforms (i.e., *Facebook*, *SoundCloud*, *Twitter*), a WoT platform (i.e., *Dweet.io*), and *ThingsNet*, our own implementation of a Level 5 STN platform (cf. Section 7.3). We validated our deployment via an *STN browser* that a human user can use to navigate and manipulate the deployed STG. In particular, this deployment validates our approach for integrating heterogenous platforms into a hypermedia-driven environment for the SWoT.

In Chapter 9, we began our discussion by introducing a multi-agent middleware whose purpose is to facilitate the development of SWoT applications. The middleware hides the details of interfacing with heterogeneous STN platforms and allows developers to focus on programming agent behavior instead. Social things are programmed as BDI agents situated in Web-enabled multi-agent environments. We then reported on our experience with using this middleware to implement the scenarios introduced in Section 5.1. In doing so, we completed our investigation by validating our approach and the foundational principles behind it.

10.2 Contributions

We summarize and group our contributions in accordance with the four research questions we address in this dissertation:

Research Question 1. *How can we bring systems of autonomous things to the Web of Things?*

We proposed *an architecture for the SWoT* (see Section 5.3) based on a well-defined set of foundational principles that provide the underpinning of our approach. This architecture conforms to the REST architectural style, which ensures unbound scalability and facilitates the integration of the large number of existing Web services. It adapts models from multi-agent research to introduce several layers of abstraction that provide a structured approach for designing and implementing systems of things that are *autonomous*, *social* (i.e., they can autonomously manage their relations and interactions with one another and with their human users), and susceptible to *regulation*. These layers of abstraction are useful to enable developers and users to cope with the envisioned complexity of the overall ecosystem.

We introduced a *multi-agent middleware for the SWoT* (see Section 9.1) that lowers the entry-barrier for the development of SWoT applications by providing developers with all the abstractions defined by our layered architecture. This middleware emphasizes minimal development overhead, and reuse and alignment with existing multi-agent technologies. It is also worth to note that from the perspective of multi-agent application developers that are not necessarily interested in the SWoT, this middleware nevertheless enables them to create multi-agent environments that can be widely distributed across heterogeneous platforms.

Research Question 2. *How can we model networks of people and autonomous things such that machines can manipulate and reason upon them?*

We introduced *a formal model for socio-technical networks (STNs)* (see Section 6.2), and in particular *digital STNs* (see Section 6.3). Following the separation of concerns principle, our digital STN model is defined in terms of *digital artifacts* and artifact-specific *operations* such that it can be easily extended and adapted to domain-specific requirements.

We introduced *a semantic description framework for STNs* based on our formal model (see Section 7.1). This framework consists of an ontology for STNs, which we call the *STN ontology* (see Section 7.1.1), and a set of guidelines that developers can use to create and publish semantic descriptions of *agents* (see Section 7.1.2), *STN platforms* (see Section 7.1.3), and *digital artifacts* (see Section 7.1.4). Following the separation of concerns principles, the STN ontology has a modular design such that it can be easily extended to fit domain- and application-specific requirements.

Research Question 3. *How can we enable things to transcend Web silos?*

We proposed *solutions to achieve uniform interfaces for heterogeneous STN platforms* (see Section 7.2). Our proposal is based on the digital STN model, which is useful to enhance interaction across heterogeneous platforms, for instance by means of *social relations* (see Definition 6.1.13) and *hosting relations* (see Definition 6.1.24). However, the same approach could be applied with other domain-specific models as well, such as the catalogues of things proposed by Blackstock et al. [Blackstock 2014a].

We proposed *a progressive, five-level strategy* for the integration of heterogenous STN platforms into a hypermedia-driven environment for the SWoT (see Section 7.3). This strategy enables developers to balance platform design and implementation autonomy against the benefits of achieving a stronger integration into the SWoT. We demonstrated that our approach and strategy supports high heterogeneity by integrating into a hypermedia-driven environment the non-hypermedia APIs of several existing social platforms, namely *Facebook*, *Twitter*, and *SoundCloud*, and a WoT platform, that is *Dweet.io*.

We introduced *STN description documents* for *Facebook*, *Twitter*, *SoundCloud* and *Dweet.io* (see Section 8.1 and Appendix A). Developers of social things can reuse these platform descriptions to enable their applications to operate *in a uniform fashion* across the mentioned platforms.

We introduced a *browser for STNs* (see Section 8.3.2), that is a Web application that can interpret STN description documents in order to interface with heterogenous STN platforms and provide users with appropriate controls to participate in those platforms. Users could use this application to browse the SWoT, and developers could use this application to test and validate the integration of their STN platforms into the SWoT.

Research Question 4. *How can we enhance discoverability and flexible interaction in the WoT?*

We proposed *an approach that applies the social network metaphor to the WoT and endows things with autonomy* (see Section 5.2.1 and Section 5.3.2). Our approach improves *discoverability* in the WoT by interconnecting people and things via explicit, typed relations. Things can then reliably use these relations to autonomously discover and interact with other agents, and they can reliably manipulate them to "rewire" themselves in a *flexible* fashion. *Flexible interaction* is also supported by STN platforms that use relations between their users to route the flow of information in STNs, therefore acting as central brokers that decouple communication. We demonstrated the validity and applicability of our approach in Chapter 9, where we reused existing multi-agent models and technologies to showcase these features via multiple application scenarios.

We introduced *a hypermedia-driven, open STN platform*, which we call ThingsNet (see Section 8.2). Developers could reuse and extend our prototype implementation to create hypermedia-driven environments that support discoverability and flexible interaction.

10.3 Future Work

The envisioned SWoT ecosystem opens up many opportunities for future research. We group some of these opportunities around topics that we present in what follows.

10.3.1 Limitations

Formalizing the Normative layer. A research challenge that we do not currently address in our work is to provide a formalization of the *Normative* layer. The normative multi-agent systems community is yet to reach consensus on many normative concepts [Andrighetto 2013]. For instance, it would be useful to have a uniform model for describing a platform's terms of use in machine-readable format. Social things could then, for instance, reason about using a platform or the other. For another example, a unified model for describing *normative organisations*. Per our discussion in Section 4.3.3, organisations are generally implemented in MAS by means of *organisation management infrastructures*, which are essentially services agents can use to join, leave and participate in organisations. If these services would expose Web APIs, then one could use a unified model of normative organisations defined in terms of organisational artifacts and artifact-specific operations to create platform descriptions very much similar to the ones we currently provide for STN platforms.

Defining a linked data protocol for STNs. Per our discussion in Section 7.2.1, we suggest the Linked Data Platform (LDP) [Speicher 2015] could serve as a standard-compliant foundation for interaction between SWoT components. However, as we noted in our discussion of the ThingsNet implementation (see Section 8.2.3), a linked data protocol for STNs would have to extend the LDP platform with domain-specific requirements.

It is also worth to note that the current LDP specifications [Speicher 2015] are based on HTTP [Fielding 2014c] and use features that cannot currently be implemented via standard CoAP features [Shelby 2014a], which supports only a subset of the features provided by HTTP. A further investigation of how the LDP could be adapted for CoAP-based implementations would be interesting in the context of the SWoT, and of the WoT in general.

Integrating constraint devices into the SWoT. In our discussion, we constantly took into consideration requirements for resource-constrained devices. Nevertheless, a more thorough investigation is necessary to explore what could be achieved. For instance, as already noted in Section 7.2.1, it would be useful to define domain-specific media types for STNs that are optimized for constrained devices.

It is also worth to note that the SWoT could benefit from non-Web IoT protocols as well. For instance, a well-known IoT protocol that currently has traction is MQTT [1], a publish/subscribe messaging protocol for machine-to-machine communication. Messages are exchanged via a central broker that handles subscription-based routing. Therefore, MQTT could be useful, for instance, to implement real-time communication between social things at the edge of the network, such as on David's STN Box in our scenarios (see Section 5.1).

10.3.2 Privacy preservation

Privacy is a well-known problem in current social platforms. Privacy concerns are not only related to large amounts of data collected by social networking services, but also to the *over-sharing of information*. Unknowingly, people often share online information about themselves or their friends without perceiving the full implications of their actions [Nissenbaum 2009]. The barriers that people are generally accustomed to in the physical world are teared down without notice in the digital

[1] http://mqtt.org/, Accessed: 06.12.2015.

world, in which information travels much faster. Furthermore, once committed, a privacy violation can not be simply reversed. We can only expect this problem to get worst in STNs with large numbers of autonomous things that are able to perceive and act on the physical-digital space, and to actively participate in the SWoT by publishing information. To address such issues, it is necessary to provide people with easy-to-use mechanisms that ensure their personal information is shared *in the right context*, *in the right ways* and *with appropriate others*.

10.3.3 Social reasoning

Searching the SWoT and relationship management. In our implementation of the "Social TV" scenario, the social TV agent follows a very simple strategy to perform an informed search in the SWoT and discover other social TVs owned by his owner's friends. Per our discussion in Section 3.2.2, searching the WoT is a well-recognized open problem. It could be fruitful, for instance, to further investigate various strategies by means of which social things could manage their relations and crawl the SWoT in a more effective manner. Such toupees are already investigated by the Social Internet of Things initiative (see Section 3.4.2).

Optimizing access to resources in the SWoT. In our discussions of the scenario implementations in Chapter 9, we have already noted that, at least for existing platforms, API rate limits can be an impediment for the development of SWoT applications. We expect that platform resources would be a constant challenge in the SWoT. Therefore, it would be useful to investigate various strategies that social things could use to share resources or even compete for resources.

Explaining actions in SWoT environments. We have suggested and argued that STNs could provide a uniform interface for remote interaction with large numbers of heterogenous, autonomous and collaborative things. Nevertheless, in our scenario implementations, social things communicate either via agent communication languages or by identifying keywords and serving hard-coded natural language responses. It would be far more useful if social things could reliably interpret natural language and also explain their actions in natural language such that they are easily understood by everyday users.

List of publications

Ciortea, A., Zimmermann, A., Boissier, O., and Florea, A. (2015). Towards a Social and Ubiquitous Web: A Model for Socio-technical Networks. In Proceedings of the 2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT), Volume 1 (pp. 461-468). IEEE.

Ciortea, A., Boissier, O., Zimmermann, A., and Florea, A. (2014). Open and Interoperable Socio-technical Networks. In Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC) (pp. 251-257). IEEE.

Ciortea, A., Boissier, O., Zimmermann, A., and Florea, A. (2013). Reconsidering the Social Web of Things: position paper. In Proceedings of the 2013 ACM International Conference on Pervasive and Ubiquitous computing adjunct publication (UbiComp) (pp. 1535-1544). ACM.

Ciortea, A., Boissier, O., Zimmermann, A., and Florea, A. (2013). Adding a Social Dimension to the Web of Things, Poster session, Journees Scientifique Systeme Embarque (SEmba).

Ciortea, A., Krupa, Y., and Vercouter, L. (2012). Designing privacy-aware social networks: A multi-agent approach. In Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics (WIMS) (pp. 8:1-8:8). ACM.

Part V

Appendices

Appendix A

Examples of STN Description Documents

In this appendix, we present the *STN description documents* (see Section 7.1.3) we have created for the platforms presented in Chapter 8, namely *Facebook*, *Twitter*, *SoundCloud*, *Dweet.io*, and *ThingsNet*.

A.1 Facebook

```
@base <http://www.facebook.com/> .
@prefix stn: <http://purl.org/stn/core#> .
@prefix stn-ops: <http://purl.org/stn/operations#> .
@prefix stn-http: <http://purl.org/stn/http#> .
@prefix http: <http://www.w3.org/2011/http#> .

<#platform>
    a stn:Platform ;
    stn:name "Facebook" ;
    stn-http:baseURL <https://graph.facebook.com/v2.4> ;
    stn-http:supportsAuth stn-http:OAuth ;
    stn-http:consumes stn-http:JSON ;
    stn-http:produces stn-http:JSON ;
    stn-ops:supports <#getAccount> ,
        <#getOutConnections> ,
        <#getInConnections> ,
        <#getGroup> ,
        <#getGroupMembers> ,
        <#getGroupsOfUser> ,
        <#getMessage> ,
        <#getReceivedMessages> ,
        <#getSentMessages> ,
        <#postMessage> ,
        <#postMessageToGroup> ,
        <#deletePost> ,
        <#getFeed> ,
        <#getUserFeed> ,
        <#getGroupFeed> .


#
# Account operations
#
```

```
<#fbAccountJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:UserAccount ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "id" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "name" ;
            stn-http:STNTerm stn:name ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "website" ;
            stn-http:STNTerm stn:swotProfile ;
        ] .

<#getAccount>
    a stn-ops:GetUserAccount ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id?fields=id,name,website" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput <#fbAccountJSONMapping> .


#
# Connection operations
#

<#fbFriendsJSONMapping>
    a stn-http:JSONArray ;
    stn-http:key "data" ;
    stn-ops:arrayOf <#fbAccountJSONMapping> .

<#getOutConnections>
    a stn-ops:GetOutgoingRelations ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/friends?fields=id,name,website" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput <#fbFriendsJSONMapping> .

<#getInConnections>
    a stn-ops:GetIncomingRelations ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/friends?fields=id,name,website" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput <#fbFriendsJSONMapping> .


#
# Group operations
#

<#fbGroupJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:Group ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "id" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "name" ;
            stn-http:STNTerm stn:name ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "about" ;
            stn-http:STNTerm stn:description ;
        ] .

<#getGroup>
    a stn-ops:GetGroup ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:GroupID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput <#fbGroupJSONMapping> .

<#getGroupMembers>
    a stn-ops:GetGroupMembers ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/members" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:GroupID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-http:key "data" ;
            stn-ops:arrayOf <#fbGroupJSONMapping> ;
        ] .

<#getGroupsOfUser>
    a stn-ops:GetGroupsOfUser ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/groups" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-http:key "data" ;
            stn-ops:arrayOf <#fbGroupJSONMapping> ;
        ] .


#
# Message operations
#

<#fbMessageJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:Message ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "id" ;
            stn-http:stnTerm stn:id ;
        ] ;
    stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "from" ;
                stn-http:stnTerm stn:hasSender ;
            ] ;
    stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "to" ;
                stn-http:stnTerm stn:hasReceiver ;
            ] ;
    stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "message" ;
                stn-http:stnTerm stn:hasBody ;
            ] .

<#getMessage>
    a stn-ops:GetMessage ;
    stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/:id" ;
            ] ;
    stn-ops:hasRequiredInput
            [ a stn-ops:MessageID ;
                stn-http:key ":id" ;
                stn-http:paramIn stn-http:Path;
            ] ;
    stn-ops:hasOutput <#fbMessageJSONMapping> .

<#getReceivedMessages>
    a stn-ops:GetReceivedMessages ;
    stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/:id/inbox" ;
            ] ;
    stn-ops:hasRequiredInput
            [ a stn-ops:UserAccountID ;
                stn-http:key ":id" ;
                stn-http:paramIn stn-http:Path;
            ] ;
    stn-ops:hasOutput
            [ a stn-http:JSONArray ;
                stn-ops:arrayOf <#fbMessageJSONMapping> ;
            ] .

<#getSentMessages>
    a stn-ops:GetSentMessages ;
    stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/:id/outbox" ;
            ] ;
    stn-ops:hasRequiredInput
```

```
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-ops:arrayOf <#fbMessageJSONMapping> ;
        ] .


#
# Feed operations
#

<#postMessage>
    a stn-ops:PostMessage ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "POST" ;
            http:requestURI "/:id/feed" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:MessageBody ;
            stn-http:key "message" ;
        ] ;
    stn-ops:hasOutput <#fbMessageJSONMapping> .

<#postMessageToGroup>
    a stn-ops:PostMessageToGroup ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "POST" ;
            http:requestURI "/:groupid/feed" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:GroupID ;
            stn-http:key ":groupid" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:MessageBody ;
            stn-http:key "message" ;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-ops:arrayOf <#fbMessageJSONMapping> ;
        ] .

<#deletePost>
    a stn-ops:DeletePost ;
    stn-ops:implementedAs
```

```
        [ a stn-http:AuthSTNRequest ;
            http:methodName "DELETE" ;
            http:requestURI "/:id" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:MessageID ;
            stn-http:key ":id" ;
        ] .

<#getFeed>
    a stn-ops:GetHomeFeed ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/home" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-ops:arrayOf <#fbMessageJSONMapping> ;
        ] .

<#getUserFeed>
    a stn-ops:GetUserAccountFeed ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/feed" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-ops:arrayOf <#fbMessageJSONMapping> ;
        ] .

<#getGroupFeed>
    a stn-ops:GetGroupFeed ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/:id/feed" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:GroupID ;
            stn-http:key ":id" ;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
```

```
                stn-ops:arrayOf <#fbMessageJSONMapping> ;
            ] .
```

Listing A.1: An STN description document for Facebook.

A.2 Twitter

```
@base <http://www.twitter.com/> .
@prefix stn: <http://purl.org/stn/core#> .
@prefix stn-ops: <http://purl.org/stn/operations#> .
@prefix stn-http: <http://purl.org/stn/http#> .
@prefix http: <http://www.w3.org/2011/http#> .

<#platform>
    a stn:Platform ;
    stn:name "Twitter" ;
    stn-http:baseURL <https://api.twitter.com/1.1> ;
    stn-http:supportsAuth stn-http:OAuth ;
    stn-ops:consumes stn-http:JSON ;
    stn-ops:produces stn-http:JSON ;
    stn-ops:supports <#getMyAccount> ,
        <#getAccount> ,
        <#getFollowers> ,
        <#getFriends> ,
        <#follow> ,
        <#unfollow> ,
        <#sendDirectMessage> ,
        <#postTweet> ,
        <#getHomeTimeline>,
        <#deleteDirectMessage> ,
        <#getDirectMessage> ,
        <#getDirectMessages> .


#
# Account operations
#

<#twitterAccountJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:UserAccount ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "screen_name" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "name" ;
            stn-http:STNTerm stn:name ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "description" ;
            stn-http:STNTerm stn:description ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "url" ;
            stn-http:STNTerm stn:swotProfile ;
        ] .

<#getMyAccount>
    a stn-ops:GetMyUserAccount ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/account/verify_credentials.json" ;
        ] ;
    stn-ops:hasOutput <#twitterAccountJSONMapping> .

<#getAccount>
    a stn-ops:GetUserAccount ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/users/show.json" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key "screen_name" ;
            stn-http:paramIn stn-http:Query;
        ] ;
    stn-ops:hasOutput <#twitterAccountJSONMapping> .


#
# Connection operations
#

<#twitterFriendsJSON>
    a stn-http:JSONArray ;
    stn-http:key "users" ;
    stn-ops:arrayOf <#twitterAccountJSONMapping> .

<#getFollowers>
    a stn-ops:GetIncomingRelations ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/followers/list.json" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key "screen_name" ;
```

```
                stn-http:paramIn stn-http:Query;
        ] ;
    stn-ops:hasOutput <#twitterFriendsJSON> .

<#getFriends>
    a stn-ops:GetOutgoingRelations ;
    stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/friends/list.json" ;
        ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:UserAccountID ;
                stn-http:key "screen_name" ;
                stn-http:paramIn stn-http:Query;
        ] ;
    stn-ops:hasOutput <#twitterFriendsJSON> .

<#follow>
    a stn-ops:CreateConnectionTo;
    stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "POST" ;
                http:requestURI "/friendships/create.json" ;
        ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:UserAccountID ;
                stn-http:key "screen_name" ;
                stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasOutput <#twitterAccountJSONMapping> .

<#unfollow>
    a stn-ops:DeleteConnectionTo ;
    stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "POST" ;
                http:requestURI "/friendships/destroy.json" ;
        ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:UserAccountID ;
                stn-http:key "screen_name" ;
                stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasOutput <#twitterAccountJSONMapping> .


#
# Direct messages and tweets
#

<#twitterDirectMessageJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:Message ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "id_str" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "sender_screen_name" ;
            stn-http:STNTerm stn:hasSender ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "receiver_screen_name" ;
            stn-http:STNTerm stn:hasReceiver ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "text" ;
            stn-http:STNTerm stn:hasBody ;
        ] .

<#tweetJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:Message ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "id_str" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "user/screen_name" ;
            stn-http:STNTerm stn:hasSender ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "text" ;
            stn-http:STNTerm stn:hasBody ;
        ] .

<#sendDirectMessage>
    a stn-ops:SendPrivateMessage;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "POST" ;
            http:requestURI "/direct_messages/new.json" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key "screen_name" ;
            stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:MessageBody ;
            stn-http:key "text" ;
            stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasOutput <#twitterDirectMessageJSONMapping> .

<#postTweet>
    a stn-ops:PostMessage;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "POST" ;
            http:requestURI "/statuses/update.json" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:MessageBody ;
            stn-http:key "status" ;
            stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasOutput <#twitterDirectMessageJSONMapping> .

<#getHomeTimeline>
    a stn-ops:GetAggregatedFeed ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/statuses/home_timeline.json" ;
        ] ;
    stn-ops:hasOutput
        [ a stn-http:JSONArray ;
            stn-ops:arrayOf <#tweetJSONMapping> ;
        ] .

<#deleteDirectMessage>
    a stn-ops:DeletePrivateMessage ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "POST" ;
            http:requestURI "/direct_messages/destroy.json" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:EntityID ;
            stn-http:key "id" ;
            stn-http:paramIn stn-http:Body;
        ] ;
    stn-ops:hasOutput <#twitterDirectMessageJSONMapping> .

<#getDirectMessage>
    a stn-ops:GetMessage ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/direct_messages/show.json" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:EntityID ;
            stn-http:key "id" ;
            stn-http:paramIn stn-http:Query;
        ] ;
    stn-ops:hasOutput <#twitterDirectMessageJSONMapping> .

<#getDirectMessages>
    a stn-ops:GetMessages ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/direct_messages.json" ;
        ] ;
    stn-ops:hasOutput [
            a stn-http:JSONArray ;
            stn-ops:arrayOf <#twitterDirectMessageJSONMapping> ;
        ] .
```

Listing A.2: An STN description document for Twitter.

A.3 SoundCloud

```
@base <http://www.soundcloud.com/> .
@prefix stn: <http://purl.org/stn/core#> .
@prefix stn-ops: <http://purl.org/stn/operations#> .
@prefix stn-http: <http://purl.org/stn/http#> .
@prefix http: <http://www.w3.org/2011/http#> .

<#platform>
    a stn:Platform ;
    stn:name "SoundCloud" ;
    stn-http:baseURL <https://api.soundcloud.com> ;
    stn-http:supportsAuth stn-http:OAuth ;
    stn-ops:consumes stn-http:JSON ;
    stn-ops:produces stn-http:JSON ;
    stn-ops:supports <#getAccount> ,
        <#getInConnections> ,
        <#getOutConnections> ,
        <#follow> ,
        <#unfollow> .


#
# Account operations
#

<#soundcloudAccountJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:UserAccount ;
    stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "id" ;
                stn-http:STNTerm stn:id ;
        ] ;
        stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "full_name" ;
                stn-http:STNTerm stn:name ;
        ] ;
        stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "description" ;
                stn-http:STNTerm stn:description ;
        ] ;
        stn-ops:contains [
                a stn-http:Mapping ;
                stn-http:key "website" ;
                stn-http:STNTerm stn:swotProfile ;
        ] .

<#getAccount>
        a stn-ops:GetUserAccount ;
        stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/users/:id" ;
        ] ;
        stn-ops:hasRequiredInput [
                a stn-ops:UserAccountID ;
                stn-http:key ":id" ;
                stn-http:paramIn stn-http:Path;
        ] ;
        stn-ops:hasOutput <#soundcloudAccountJSONMapping> .


#
# Connection operations
#

<#soundcloudFriendsJSON>
        a stn-http:JSONArray ;
        stn-ops:arrayOf <#soundcloudAccountJSONMapping> .

<#getInConnections>
        a stn-ops:GetIncomingRelations ;
        stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/users/:id/followers" ;
        ] ;
        stn-ops:hasRequiredInput [
                a stn-ops:UserAccountID ;
                stn-http:key ":id" ;
                stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput <#soundcloudFriendsJSON> .

<#getOutConnections>
    a stn-ops:GetOutgoingRelations ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/users/:id/followings" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key ":id" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput <#soundcloudFriendsJSON> .

<#follow>
    a stn-ops:CreateRelationTo;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "PUT" ;
            http:requestURI "/users/:id1/followings/:id2" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key ":id1" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:TargetUserAccountID ;
            stn-http:key ":id2" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput <#soundcloudAccountJSONMapping> .

<#unfollow>
    a stn-ops:DeleteRelationTo ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "DELETE" ;
            http:requestURI "/users/:id1/followings/:id2" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:UserAccountID ;
            stn-http:key ":id1" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:TargetUserAccountID ;
            stn-http:key ":id2" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput <#soundcloudAccountJSONMapping> .
```

Listing A.3: An STN description document for SoundCloud.

A.4 Dweet.io

```
@base <http://www.dweet.io/> .
@prefix stn: <http://purl.org/stn/core#> .
@prefix stn-ops: <http://purl.org/stn/operations#> .
@prefix stn-http: <http://purl.org/stn/http#> .
@prefix http: <http://www.w3.org/2011/http#> .

<#platform>
    a stn:Platform ;
    stn:name "dweet.io" ;
    stn-http:baseURL <https://dweet.io> ;
    stn-ops:consumes stn-http:JSON ;
    stn-ops:produces stn-http:JSON ;
    stn-ops:supports <#createAccount> ,
        <#postDweet> ,
        <#getDweets> .


<#dweetJSONMapping>
    a stn-ops:Representation ;
    stn-ops:mediaType stn-http:JSON ;
    stn-ops:entityType stn:DataObject ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "created" ;
            stn-http:STNTerm stn:id ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "thing" ;
            stn-http:STNTerm stn:createdBy ;
        ] ;
    stn-ops:contains [
            a stn-http:Mapping ;
            stn-http:key "content" ;
            stn-http:STNTerm stn:data ;
        ] .

<#createAccount>
    a stn-ops:CreateUserAccount ;
    stn-ops:implementedAs
        [ a stn-http:STNRequest ;
            http:methodName "GET" ;
            http:requestURI "/dweet/for/:accountId" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-http:UserAccountID ;
```

```
                stn-http:key ":accountId" ;
                stn-http:paramIn stn-http:Path ;
            ] ;
    stn-ops:hasOutput
            [ a stn-ops:Representation ;
                stn-ops:mediaType stn-http:JSON ;
                stn-ops:entityType stn:UserAccount ;
                stn-http:key "with" ;
                stn-ops:contains [
                        a stn-http:Mapping ;
                        stn-http:key "thing" ;
                        stn-http:STNTerm stn:createdBy;
                    ] ;
            ] .

<#postDweet>
    a stn-ops:PostDataToUserAccountFeed ;
    stn-ops:implementedAs [
                a stn-http:STNRequest ;
                http:methodName "POST" ;
                http:requestURI "/dweet/for/:accountId" ;
            ] ;
    stn-ops:hasRequiredInput [
                a stn-http:UserAccountID ;
                stn-http:key ":accountId" ;
                stn-http:paramIn stn-http:Path ;
            ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:Representation ;
                stn-ops:mediaType stn-http:JSON ;
                stn-ops:entityType stn:DataObject ;
                stn-http:paramIn stn-http:Body;
            ] ;
    stn-ops:hasOutput [
                a stn-ops:Representation ;
                stn-http:key "with" ;
                stn-ops:representationOf <#dweetJSONMapping> ;
            ] .

<#getDweets>
    a stn-ops:GetUserAccountFeed;
    stn-ops:implementedAs [
                a stn-http:STNRequest ;
                http:methodName "GET" ;
                http:requestURI "/get/dweets/for/:accountId" ;
            ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:UserAccountID ;
                stn-http:key ":accountId" ;
                stn-http:paramIn stn-http:Path ;
            ] ;
    stn-ops:hasOutput [
                a stn-http:JSONArray ;
                stn-http:key "with" ;
                stn-ops:arrayOf <#dweetJSONMapping> ;
            ] .
```

Listing A.4: An STN description document for Dweet.io.

A.5 ThingsNet

```
@base <http://www.thingsnet.com/> .
@prefix stn: <http://purl.org/stn/core#> .
@prefix stn-ops: <http://purl.org/stn/operations#> .
@prefix stn-http: <http://purl.org/stn/http#> .
@prefix http: <http://www.w3.org/2011/http#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix format: <http://www.w3.org/ns/formats/> .

<#platform>
    a stn:STNPlatform ;
    stn:name "ThingsNet" ;
    stn-http:baseURL <http://localhost:9000> ;
    stn-http:supportsAuth stn-http:WebID ;
    stn-http:consumes format:Turtle ;
    stn-http:produces format:Turtle ;
    stn-ops:supports <#createAccount> ,
        <#getAccount> ,
        <#getAccountForAgent> ,
        <#deleteAccount> ,
        <#createConnectionTo> ,
        <#getOutConnections> ,
        <#getInConnections> ,
        <#deleteConnectionTo> ,
        <#sendMessage> ,
        <#getMessage> ,
        <#getMessages> ,
        <#deleteMessage> .



#
# Account operations
#

<#createAccount>
    a stn-ops:CreateUserAccount ;
    stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "POST" ;
                http:requestURI "/users" ;
            ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:SocialThingClass ;
            ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:SocialThingOwner ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:DisplayedName ;
        ] ;
    stn-ops:hasInput [
            a stn-ops:Description ;
        ] ;
    stn-ops:hasOutput [
            a stn-ops:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:entityType stn:UserAccount ;
        ] .

<#getAccount>
    a stn-ops:GetUserAccount ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI ":accountUri" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:UserAccountURI ;
            stn-http:key ":accountUri" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput
        [ a stn-ops:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:entityType stn:UserAccount ;
        ] .

<#getAccountForAgent>
    a stn-ops:WhoIsAgent ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/users" ;
        ] ;
    stn-ops:hasRequiredInput [
            a stn-ops:AgentURI ;
            stn-http:key "agentUri" ;
            stn-http:paramIn stn-http:Query ;
        ] ;
    stn-ops:hasOutput [
            a stn-ops:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:entityType stn:UserAccount ;
        ] .

<#deleteAccount>
    a stn-ops:DeleteUserAccount ;
    stn-ops:implementedAs [
            a stn-http:AuthSTNRequest ;
```

```
                http:methodName "DELETE" ;
                http:requestURI "/users" ;
            ] ;
    stn-ops:hasOutput [
                a stn-ops:Representation ;
                stn-ops:mediaType format:Turtle ;
                stn-ops:entityType stn:UserAccount ;
            ] .



#
# Connection operations
#

<#createConnectionTo>
    a stn-ops:CreateRelationTo;
    stn-ops:implementedAs [
                a stn-http:AuthSTNRequest ;
                http:methodName "POST" ;
                http:requestURI "/connections" ;
            ] ;
    stn-ops:hasRequiredInput [
                a stn-ops:AgentURI ;
            ] ;
    stn-ops:hasOutput [
                a stn-ops:Representation ;
                stn-ops:mediaType format:Turtle ;
                stn-ops:entityType stn:UserAccount ;
            ] .

<#getOutConnections>
    a stn-ops:GetOutgoingRelations ;
    stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/connections/out" ;
            ] ;
    stn-ops:hasRequiredInput
            [ a stn-ops:UserAccountURI ;
                stn-http:key "accountUri" ;
                stn-http:paramIn stn-http:Query ;
            ] ;
    stn-ops:hasOutput
            [ a stn-ops:Representation ;
                stn-ops:mediaType format:Turtle ;
                stn-ops:arrayOf stn:UserAccount ;
            ] .

<#getInConnections>
    a stn-ops:GetIncomingRelations ;
    stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "GET" ;
                http:requestURI "/connections/in" ;
            ] ;
        stn-ops:hasRequiredInput
            [ a stn-ops:UserAccountURI ;
                stn-http:key "accountUri" ;
                stn-http:paramIn stn-http:Query ;
            ] ;
        stn-ops:hasOutput
            [ a stn-ops:Representation ;
                stn-ops:mediaType format:Turtle ;
                stn-ops:arrayOf stn:UserAccount ;
            ] .

<#deleteConnectionTo>
        a stn-ops:DeleteRelationTo ;
        stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "DELETE" ;
                http:requestURI "/connections" ;
            ] ;
        stn-ops:hasRequiredInput [
                a stn-ops:AgentURI ;
                stn-http:key "agentUri" ;
                stn-http:paramIn stn-http:Query ;
            ] ;
        stn-ops:hasOutput
            [ a stn-ops:Representation ;
                stn-ops:mediaType format:Turtle ;
                stn-ops:entityType stn:UserAccount ;
            ] .



#
# Message operations
#

<#sendMessage>
        a stn-ops:SendMessage ;
        stn-ops:implementedAs
            [ a stn-http:AuthSTNRequest ;
                http:methodName "POST" ;
                http:requestURI "/messages" ;
            ] ;
        stn-ops:hasRequiredInput [
                a stn-ops:UserAccountURI ;
            ] ;
        stn-ops:hasInput [
                a stn-ops:MessageURI ;
            ] ;
        stn-ops:hasInput [
                a stn-ops:Subject ;
            ] ;
        stn-ops:hasInput [
            a stn-ops:MessageBody ;
        ] ;
    stn-ops:hasOutput [
            a stn-ops:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:entityType stn:Message ;
        ] .

<#deleteMessage>
    a stn-ops:DeletePrivateMessage ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "DELETE" ;
            http:requestURI ":messageUri" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:MessageURI ;
            stn-http:key "messageUri" ;
            stn-http:paramIn stn-http:Path ;
        ] .

<#getMessage>
    a stn-ops:GetMessage ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI ":messageUri" ;
        ] ;
    stn-ops:hasRequiredInput
        [ a stn-ops:MessageURI ;
            stn-http:key "messageUri" ;
            stn-http:paramIn stn-http:Path ;
        ] ;
    stn-ops:hasOutput
        [ a stn-ops:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:entityType stn:Message ;
        ] .

<#getMessages>
    a stn-ops:GetMessages ;
    stn-ops:implementedAs
        [ a stn-http:AuthSTNRequest ;
            http:methodName "GET" ;
            http:requestURI "/messages" ;
        ] ;
    stn-ops:hasOutput
        [ a stn-ops:Representation ;
            stn-ops:mediaType format:Turtle ;
            stn-ops:arrayOf stn:Message ;
        ] .
```

Listing A.5: An STN description document for ThingsNet.

Appendix B

Normative Organizations for Home Automation

In this appendix, we present the complete specification of the MOISE+ organisation [Hubner 2007] we have created to implement the "A welcoming home" scenario (see Section 5.1.4). Our implementation is discussed in detail in Section 9.5.

The organisational specification is shown in Listing B.1. It defines the *structural*, *functional* and *normative* dimensions of an organisaton for David's *social things*.

The *structural specification* defines the roles available within the organisation, links among those roles, and a group for David's *social things*.

The *functional specification* defines a scheme to achieve the goal of setting up David's home for his arrival. This goal is decomposed in multiple subgoals grouped in missions.

The *normative specification* defines five *norms* that assign *missions* defined by the functional specification to *roles* defined by the structural specification by means of *obligations*.

```
<?xml version="1.0" encoding="UTF-8"?>

<?xml-stylesheet href="http://moise.sourceforge.net/xml/os.xsl" type="
    text/xsl" ?>

<organisational-specification
     id="home"
     os-version="0.8"

     xmlns='http://moise.sourceforge.net/os'
     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
     xsi:schemaLocation='http://moise.sourceforge.net/os
                          http://moise.sourceforge.net/xml/os.xsd' >

<structural-specification>

<role-definitions>
 <role id="social_thing" />
 <role id="car" > <extends role="social_thing"/> </role>
 <role id="vacuum_cleaner" > <extends role="social_thing"/> </role>
 <role id="thermostat" > <extends role="social_thing"/> </role>
 <role id="light_manager" > <extends role="social_thing"/> </role>
 <role id="curtains" > <extends role="light_manager"/> </role>
 <role id="lights" > <extends role="light_manager"/>  </role>
 <role id="speakers" > <extends role="social_thing"/>  </role>
</role-definitions>

<group-specification id="home_group">
 <roles>
  <role id="car" min="1"/>
  <role id="vacuum_cleaner" min="1"/>
  <role id="thermostat" min="1"/>
  <role id="light_manager" min="1"/>
  <role id="speakers" min="1"/>
 </roles>

 <links>
  <link from="social_thing" to="social_thing" type="communication"
      scope="intra-group" />
 </links>
</group-specification>
</structural-specification>

<functional-specification>
 <scheme id="welcome_home_scheme">
   <goal id="warm_welcoming">
     <plan operator="sequence">
       <goal id="announce_departure"/>
       <goal id="prepare_home_ambient">
         <plan operator="parallel">
           <goal id="vacuum_house"    ttf="20 minutes" ds="Vacuum the
               house."/>
           <goal id="set_ambient_temperature"    ttf="20 minutes" ds="
               Prepare a warm house, literally."/>
           <goal id="prepare_arrival">
             <plan operator="sequence">
               <goal id="announce_arrival"/>
               <goal id="make_last_preperations">
                 <plan operator="parallel">
                   <goal id="set_ambient_lighting"/>
                   <goal id="set_ambient_music"/>
                 </plan>
               </goal>
             </plan>
           </goal>
         </plan>
       </goal>
     </plan>
   </goal>

   <!-- <mission id="m1" min="1" max="1">
     <goal id="warm_welcoming"/>
     <goal id="announce_departure"/>
     <goal id="prepare_home_ambient"/>
     <goal id="prepare_arrival"/>
     <goal id="announce_arrival"/>
     <goal id="make_last_preperations"/>
```

```
    </mission> -->

    <mission id="m1" min="1" max="1">
      <goal id="announce_departure"/>
      <goal id="announce_arrival"/>
      <goal id="warm_welcoming"/>
    </mission>

    <mission id="m2" min="1" max="1">
      <goal id="vacuum_house"/>
    </mission>

    <mission id="m3" min="1" max="1">
      <goal id="set_ambient_temperature"/>
    </mission>

    <mission id="m4" min="1">
      <goal id="set_ambient_lighting"/>
    </mission>

    <mission id="m5" min="1" max="1">
      <goal id="set_ambient_music"/>
    </mission>
 </scheme>
</functional-specification>

<normative-specification>
 <norm id="norm1" type="obligation" role="car" mission="m1"/>
 <norm id="norm2" type="obligation" role="vacuum_cleaner" mission="m2"
     />
 <norm id="norm3" type="obligation" role="thermostat" mission="m3"/>
 <norm id="norm4" type="obligation" role="light_manager" mission="m4"/>
 <norm id="norm5" type="obligation" role="speakers" mission="m5"/>
</normative-specification>

</organisational-specification>
```

Listing B.1: A MOISE+ organisation for automating David's home.

Bibliography

[Andrighetto 2013] Giulia Andrighetto, Guido Governatori, Pablo Noriega and Leendert WN van der Torre. Normative multi-agent systems. Schloss Dagstuhl-Leibniz-Zentrum für Informatik GmbH, 2013. (Cited on pages 51 and 181.)

[Argente 2013] Estefanía Argente, Olivier Boissier, Sergio Esparcia, Jana Görmer, Kristi Kirikal and Kuldar Taveter. *Describing Agent Organisations.* In Agreement Technologies, pages 253–275. Springer, 2013. (Cited on pages 47 and 52.)

[Armbrust 2010] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica*et al. A view of cloud computing.* Communications of the ACM, vol. 53, no. 4, pages 50–58, 2010. (Cited on page 82.)

[Asl 2013] Hamid Zargari Asl, Antonio Iera, Luigi Atzori and Giacomo Morabito. *How often social objects meet each other? Analysis of the properties of a social network of IoT devices based on real data.* In Global Communications Conference (GLOBECOM), 2013 IEEE, pages 2804–2809. IEEE, 2013. (Cited on page 41.)

[Atzori 2010] Luigi Atzori, Antonio Iera and Giacomo Morabito. *The internet of things: A survey.* Computer networks, vol. 54, no. 15, pages 2787–2805, 2010. (Cited on page 23.)

[Atzori 2012] Luigi Atzori, Antonio Iera, Giacomo Morabito and Michele Nitti. *The social internet of things (siot)–when social networks meet the internet of things: Concept, architecture and network characterization.* Computer Networks, vol. 56, no. 16, pages 3594–3608, 2012. (Cited on pages 39, 41 and 42.)

[Austin 1962] John L Austin. *How to do things with words*, 1962. (Cited on page 48.)

[Balke 2013] Tina Balke, Célia da Costa Pereira, Frank Dignum, Emiliano Lorini, Antonino Rotolo, Wamberto Vasconcelos and Serena Villata. *Norms in MAS: Definitions and Related Concepts.* Normative Multi-Agent Systems, vol. 4, pages 1–31, 2013. (Cited on page 51.)

[Beckett 2008] David Beckett and Tim Berners-Lee. *Turtle - Terse RDF Triple Language, W3C Team Submission 14 January 2008.* W3C team submission, World Wide Web Consortium (W3C), January 14 2008. (Cited on pages 32, 103, 106 and 112.)

[Berners-Lee 2001] Tim Berners-Lee, James Hendler, Ora Lassila*et al. The semantic web.* Scientific american, vol. 284, no. 5, pages 28–37, 2001. (Cited on page 20.)

[Berners-Lee 2005] T. Berners-Lee, R. Fielding and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax.* RFC 3986 (INTERNET STANDARD), January 2005. Updated by RFCs 6874, 7320. (Cited on pages 11, 111 and 116.)

[Berners-Lee 2006] Tim Berners-Lee. *Linked data-design issues.* 2006. (Cited on page 21.)

[Berners-Lee 2009] T Berners-Lee. *Socially aware cloud storage.* W3C Design Note http://www.w3.org/DesignIssues/CloudStorage.html, 2009. (Cited on pages 16 and 17.)

[Berthet 1992] Sabine Berthet, Yves Demazeau and Oliver Boissier. *Knowing each other better.* In Proceedings of the 11th International Workshop on Distributed Artificial Intelligence, pages 23–42. Glen Arbor USA, 1992. (Cited on page 50.)

[Blackstock 2010] Michael Blackstock, Nima Kaviani, Rodger Lea and Adrian Friday. *MAGIC Broker 2: An open and extensible platform for the Internet of Things.* In Internet of Things (IOT), 2010, pages 1–8. IEEE, 2010. (Cited on page 39.)

[Blackstock 2011] Michael Blackstock, Rodger Lea and Adrian Friday. *Uniting online social networks with places and things.* In Proceedings of the Second International Workshop on Web of Things, page 5. ACM, 2011. (Cited on pages 35 and 40.)

[Blackstock 2012] Michael Blackstock and Rodger Lea. *IoT mashups with the WoTKit.* In Internet of Things (IOT), 2012 3rd International Conference on the, pages 159–166. IEEE, 2012. (Cited on pages 2, 38 and 39.)

[Blackstock 2013] Michael Blackstock and Rodger Lea. *Toward interoperability in a web of things.* In Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication, pages 1565–1574. ACM, 2013. (Cited on pages 30 and 31.)

[Blackstock 2014a] Michael Blackstock and Rodger Lea. *IoT interoperability: A hub-based approach.* In Internet of Things (IOT), 2014 International Conference on the, pages 79–84. IEEE, 2014. (Cited on pages 1, 28, 30, 31, 32 and 180.)

[Blackstock 2014b] Michael Blackstock and Rodger Lea. *Towards a distributed data flow platform for the web of things.* 2014. (Cited on page 38.)

[Blumenthal 2001] Marjory S Blumenthal and David D Clark. *Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world.* ACM Transactions on Internet Technology (TOIT), vol. 1, no. 1, pages 70–109, 2001. (Cited on page 25.)

[Boissier 2007] Olivier Boissier, Jomi Fred Hübner and Jaime Simão Sichman. *Organization oriented programming: From closed to open organizations.* In Engineering Societies in the Agents World VII, pages 86–105. Springer, 2007. (Cited on page 47.)

[Boissier 2013] Olivier Boissier, Rafael H Bordini, Jomi F Hübner, Alessandro Ricci and Andrea Santi. *Multi-agent oriented programming with JaCaMo.* Science of Computer Programming, vol. 78, no. 6, pages 747–761, 2013. (Cited on pages 46, 47, 48, 73, 154 and 172.)

[Bojars 2010] Uldis Bojars and John G. Breslin. *SIOC Core Ontology Specification.* `http://rdfs.org/sioc/spec/`, March 2010. Accessed: 2015-02-09. (Cited on pages 18 and 100.)

[Bonomi 2012] Flavio Bonomi, Rodolfo Milito, Jiang Zhu and Sateesh 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, pages 13–16. ACM, 2012. (Cited on page 82.)

[Bordini 2005] Rafael H Bordini, Mehdi Dastani, Jürgen Dix and A El Fallah Seghrouchni. Multi-agent programming. Springer, 2005. (Cited on page 47.)

[Bordini 2006] Rafael H Bordini and Jomi F Hübner. *BDI agent programming in AgentSpeak using Jason.* In Computational logic in multi-agent systems, pages 143–164. Springer, 2006. (Cited on page 155.)

[Bordini 2007] Rafael H Bordini, Jomi Fred Hübner and Michael Wooldridge. Programming multi-agent systems in agentspeak using jason, volume 8. John Wiley & Sons, 2007. (Cited on pages 46, 49, 73, 154, 155, 156, 158, 160, 163, 164 and 165.)

[Bormann 2013] C. Bormann and P. Hoffman. *Concise Binary Object Representation (CBOR).* RFC 7049 (Proposed Standard), October 2013. (Cited on page 112.)

[Bormann 2014] C. Bormann, M. Ersue and A. Keranen. *Terminology for Constrained-Node Networks.* RFC 7228 (Informational), May 2014. (Cited on page 25.)

[Bratman 1988] Michael E. Bratman, David Israel and Martha E. Pollack. *Plans and resource-bounded practical reasoning.* Computational intelligence, vol. 4, no. 4, pages 349–355, 1988. (Cited on page 155.)

[Bray 2014] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format.* RFC 7159 (Proposed Standard), March 2014. (Cited on pages 12, 32, 109, 112, 117, 128 and 133.)

[Breslin 2009] John Breslin, Alexandre Passant and Stefan Decker. The social semantic web. Springer Science & Business Media, 2009. (Cited on page 20.)

[Brickley 2004] Dan Brickley and Ramanathan V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004*. W3C Recommendation, World Wide Web Consortium (W3C), February 10 2004. (Cited on page 21.)

[Brickley 2014] Dan Brickley and Libby Miller. *FOAF vocabulary specification 0.99*. `http://xmlns.com/foaf/spec/`, January 2014. Accessed: 2015-02-09. (Cited on pages 18, 83, 100, 105 and 149.)

[Broll 2009] Gregor Broll, Enrico Rukzio, Massimo Paolucci, Matthias Wagner, Albrecht Schmidt and Heinrich Hussmann. *Perci: Pervasive service interaction with the internet of things*. Internet Computing, IEEE, vol. 13, no. 6, pages 74–81, 2009. (Cited on pages 33 and 34.)

[Bruber 1993] T Bruber. *A translation approach to portable ontology specification*. Knowledge Acquisition, vol. 5, pages 199–220, 1993. (Cited on page 20.)

[Brush 2011] AJ Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu and Colin Dixon. *Home automation in the wild: challenges and opportunities*. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 2115–2124. ACM, 2011. (Cited on pages 2 and 33.)

[Carabelea 2005] Cosmin Carabelea, Olivier Boissier and Cristiano Castelfranchi. *Using social power to enable agents to reason about being part of a group*. In Engineering Societies in the Agents World V, pages 166–177. Springer, 2005. (Cited on page 50.)

[Castelfranchi 1992] Cristiano Castelfranchi, Maria Miceli and Amedeo Cesta. *Dependence relations among autonomous agents*. Decentralized AI, vol. 3, pages 215–227, 1992. (Cited on page 50.)

[Ciortea 2012] Andrei Ciortea, Yann Krupa and Laurent Vercouter. *Designing privacy-aware social networks: A multi-agent approach*. In 2nd International Conference on Web Intelligence, Mining and Semantics, WIMS '12, Craiova, Romania, June 6-8, 2012, pages 8:1–8:8. ACM, 2012. (Cited on page 73.)

[Colistra 2014] Giuseppe Colistra, Virginia Pilloni and Luigi Atzori. *The problem of task allocation in the Internet of Things and the consensus-based approach*. Computer Networks, vol. 73, pages 98–111, 2014. (Cited on page 41.)

[Cooper 2008] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley and W. Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280 (Proposed Standard), May 2008. Updated by RFC 6818. (Cited on page 17.)

[Coutinho 2005] Luciano R Coutinho, Jaime S Sichman, Olivier Boissier*et al. Modeling organization in mas: A comparison of models*. In First Workshop on

Software Engineering for Agent-oriented Systems, pages 1–10, 2005. (Cited on page 53.)

[Cyganiak 2014] Richard Cyganiak, David Wood and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014.* W3C Recommendation, World Wide Web Consortium (W3C), February 25 2014. (Cited on pages 20 and 21.)

[Dastani 2008] Mehdi Dastani. *2APL: a practical agent programming language.* Autonomous agents and multi-agent systems, vol. 16, no. 3, pages 214–248, 2008. (Cited on page 46.)

[Davis 1983] Randall Davis and Reid G Smith. *Negotiation as a metaphor for distributed problem solving.* Artificial intelligence, vol. 20, no. 1, pages 63–109, 1983. (Cited on page 49.)

[Demazeau 1995] Yves Demazeau. *From interactions to collective behaviour in agent-based systems.* In In: Proceedings of the 1st. European Conference on Cognitive Science. Saint-Malo. Citeseer, 1995. (Cited on pages 46 and 47.)

[Derthick 2013] Katie Derthick, James Scott, Nicolas Villar and Christian Winkler. *Exploring smartphone-based web user interfaces for appliances.* In Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services, pages 227–236. ACM, 2013. (Cited on pages 33 and 34.)

[Dignum 2005] Virginia Dignum, Javier Vázquez-Salceda and Frank Dignum. *Omni: Introducing social structure, norms and ontologies into agent organizations.* In Programming multi-agent systems, pages 181–198. Springer, 2005. (Cited on page 52.)

[Dunkels 2009] Adam Dunkels*et al.* *Efficient application integration in IP-based sensor networks.* In Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings, pages 43–48. ACM, 2009. (Cited on page 25.)

[Dusseault 2010] L. Dusseault and J. Snell. *PATCH Method for HTTP.* RFC 5789 (Proposed Standard), March 2010. (Cited on pages 21, 22 and 94.)

[Esteva 2002] Marc Esteva, David De La Cruz and Carles Sierra. *ISLANDER: an electronic institutions editor.* In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3, pages 1045–1052. ACM, 2002. (Cited on page 52.)

[Ferber 1998] Jacques Ferber and Olivier Gutknecht. *A meta-model for the analysis and design of organizations in multi-agent systems.* In Multi Agent Systems, 1998. Proceedings. International Conference on, pages 128–135. IEEE, 1998. (Cited on page 52.)

[Fielding 2000] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000. (Cited on page 10.)

[Fielding 2008] Roy T Fielding. *REST APIs must be hypertext-driven*, October 2008. [Online; posted 20-October-2008]. (Cited on page 12.)

[Fielding 2014a] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Authentication.* RFC 7235 (Proposed Standard), June 2014. (Cited on pages 16, 104 and 116.)

[Fielding 2014b] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing.* RFC 7230 (Proposed Standard), June 2014. (Cited on page 21.)

[Fielding 2014c] R. Fielding and J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* RFC 7231 (Proposed Standard), June 2014. (Cited on pages 11, 12, 21, 22, 94, 112, 114 and 182.)

[Finin 1995] Tim Finin, Yannis Labrou and James Mayfield. *KQML as an agent communication language.* 1995. (Cited on page 49.)

[Formo 2012] Joakim Formo. *A Social Web of Things*, 2012. (Cited on pages 2, 33, 35 and 39.)

[Fornara 2002] Nicoletta Fornara and Marco Colombetti. *Operational specification of a commitment-based agent communication language.* In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2, pages 536–542. ACM, 2002. (Cited on page 49.)

[Gregorio 2007] J. Gregorio and B. de hOra. *The Atom Publishing Protocol.* RFC 5023 (Proposed Standard), October 2007. (Cited on page 21.)

[Guinard 2009] Dominique Guinard, Vlad Trifa, Thomas Pham and Olivier Liechti. *Towards physical mashups in the web of things.* In Networked Sensing Systems (INSS), 2009 Sixth International Conference on, pages 1–4. IEEE, 2009. (Cited on pages 1 and 37.)

[Guinard 2010a] Dominique Guinard, Mathias Fischer and Vlad Trifa. *Sharing using social networks in a composable web of things.* In Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on, pages 702–707. IEEE, 2010. (Cited on page 39.)

[Guinard 2010b] Dominique Guinard, Vlad Trifa and Erik Wilde. *A resource oriented architecture for the web of things.* In Internet of Things (IOT), 2010, pages 1–8. IEEE, 2010. (Cited on pages 1, 2, 23, 24 and 25.)

[Guinard 2011a] Dominique Guinard. *A Web of things application architecture.* PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 19891, 2011, 2011. (Cited on pages 23, 24, 36 and 38.)

[Guinard 2011b] Dominique Guinard, Christian Floerkemeier and Sanjay Sarma. *Cloud computing, REST and mashups to simplify RFID application development and deployment.* In Proceedings of the Second International Workshop on Web of Things, page 9. ACM, 2011. (Cited on page 38.)

[Halpin 2010] Harry Halpin and Mischa Tuffield. *A Standards-based, Open and Privacy-aware Social Web.* W3C Social Web Incubator Group Report 6th December, 2010. (Cited on pages 2, 13, 14, 15, 16, 17 and 18.)

[Hammer-Lahav 2010] E. Hammer-Lahav. *The OAuth 1.0 Protocol.* RFC 5849 (Informational), April 2010. Obsoleted by RFC 6749. (Cited on pages 134 and 140.)

[Hardt 2012] D. Hardt. *The OAuth 2.0 Authorization Framework.* RFC 6749 (Proposed Standard), October 2012. (Cited on pages 16, 129, 132 and 140.)

[Hardy 2010] Robert Hardy, Enrico Rukzio, Paul Holleis and Matthias Wagner. *Mobile interaction with static and dynamic NFC-based displays.* In Proceedings of the 12th international conference on Human computer interaction with mobile devices and services, pages 123–132. ACM, 2010. (Cited on page 34.)

[Hartke 2015] K. Hartke. *Observing Resources in the Constrained Application Protocol (CoAP).* RFC 7641 (Proposed Standard), September 2015. (Cited on page 113.)

[Holmquist 2001] Lars Erik Holmquist, Friedemann Mattern, Bernt Schiele, Petteri Alahuhta, Michael Beigl and Hans-W Gellersen. *Smart-its friends: A technique for users to easily establish connections between smart artefacts.* In Ubicomp 2001: Ubiquitous Computing, pages 116–122. Springer, 2001. (Cited on pages 34 and 40.)

[Hubner 2007] Jomi F Hubner, Jaime S Sichman and Olivier Boissier. *Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels.* International Journal of Agent-Oriented Software Engineering, vol. 1, no. 3-4, pages 370–395, 2007. (Cited on pages 52, 154, 171, 172 and 211.)

[Huhns 2001] Michael N Huhns. *Interaction-oriented programming.* In Agent-Oriented Software Engineering, pages 29–44. Springer, 2001. (Cited on page 47.)

[Hui 2008] Jonathan W Hui and David E Culler. *IP is dead, long live IP for wireless sensor networks.* In Proceedings of the 6th ACM conference on Embedded network sensor systems, pages 15–28. ACM, 2008. (Cited on page 25.)

[Hui 2011] J. Hui and P. Thubert. *Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks.* RFC 6282 (Proposed Standard), September 2011. (Cited on page 1.)

[Ishaq 2013] Isam Ishaq, David Carels, Girum K Teklemariam, Jeroen Hoebeke, Floris Van den Abeele, Eli De Poorter, Ingrid Moerman and Piet Demeester. *IETF standardization in the field of the internet of things (IoT): a survey.* Journal of Sensor and Actuator Networks, vol. 2, no. 2, pages 235–287, 2013. (Cited on pages 1 and 30.)

[Jacobs 2004] Ian Jacobs and Norman Walsh. *Architecture of the World Wide Web, Volume One, W3C Recommendation 15 December 2004.* W3C Recommendation, World Wide Web Consortium (W3C), December 15 2004. (Cited on pages 10 and 21.)

[Jennings 1998] Nicholas R Jennings and Michael Wooldridge. *Applications of intelligent agents.* In Agent technology, pages 3–28. Springer, 1998. (Cited on page 46.)

[Jøsang 2007] Audun Jøsang, Roslan Ismail and Colin Boyd. *A survey of trust and reputation systems for online service provision.* Decision support systems, vol. 43, no. 2, pages 618–644, 2007. (Cited on page 52.)

[Jøsang 2009] Audun Jøsang and Jennifer Golbeck. *Challenges for robust trust and reputation systems.* In Proceedings of the 5th International Workshop on Security and Trust Management (SMT 2009), Saint Malo, France, 2009. (Cited on page 52.)

[Kindberg 2002] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris*et al. People, places, things: Web presence for the real world.* Mobile Networks and Applications, vol. 7, no. 5, pages 365–376, 2002. (Cited on pages 23 and 35.)

[Kleinfeld 2014] Robert Kleinfeld, Lukasz Radziwonowicz and Charalampos Doukas. *glue. things–a Mashup Platform for wiring the Internet of Things with the Internet of Services.* 2014. (Cited on pages 2 and 38.)

[Koch 2011] Johannes Koch, Carlos A Velasco and Philip Ackermann. *HTTP Vocabulary in RDF 1.0.* `http://www.w3.org/TR/HTTP-in-RDF10/`, May 2011. Accessed: 2015-09-05. (Cited on pages 100 and 104.)

[Kortuem 2010] Gerd Kortuem, Fahim Kawsar, Daniel Fitton and Vasughi Sundramoorthy. *Smart objects as building blocks for the internet of things.* Internet Computing, IEEE, vol. 14, no. 1, pages 44–51, 2010. (Cited on page 40.)

[Kovatsch 2015] Frank Matthias Kovatsch. *Scalable Web Technology for the Internet of Things*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22398, 2015. (Cited on pages 1 and 25.)

[Kranz 2010a] Matthias Kranz, Paul Holleis and Albrecht Schmidt. *Embedded interaction: Interacting with the internet of things*. Internet Computing, IEEE, vol. 14, no. 2, pages 46–53, 2010. (Cited on pages 33 and 34.)

[Kranz 2010b] Matthias Kranz, Luis Roalter and Florian Michahelles. *Things that twitter: social networks and the internet of things*. In What can the Internet of Things do for the Citizen (CIoT) Workshop at The Eighth International Conference on Pervasive Computing (Pervasive 2010), pages 1–10, 2010. (Cited on page 39.)

[Krupa 2012] Yann Krupa and Laurent Vercouter. *Handling privacy as contextual integrity in decentralized virtual communities: The PrivaCIAS framework*. Web Intelligence and Agent Systems, vol. 10, no. 1, pages 105–116, 2012. (Cited on page 73.)

[Labrou 1994] Yannis Labrou and Tim Finin. *A semantics approach to KQML – a general purpose communication language for software agents*. In Proceedings of the third international conference on Information and knowledge management, pages 447–455. ACM, 1994. (Cited on page 49.)

[Labrou 1999] Yannis Labrou, Tim Finin and Yun Peng. *Agent communication languages: The current landscape*. IEEE Intelligent systems, vol. 14, no. 2, pages 45–52, 1999. (Cited on pages 46 and 49.)

[Langheinrich 2000] Marc Langheinrich, Friedemann Mattern, Kay Römer and Harald Vogt. *First steps towards an event-based infrastructure for smart things*. In Ubiquitous Computing Workshop (PACT 2000), 2000. (Cited on page 34.)

[Lardinois 2015] Frederic Lardinois. *The Parrot Pot And H2O Give You A Robotic Green Thumb*, January 2015. [Online; posted 4-January-2015]. (Cited on page 34.)

[Lieberman 2007] Joshua Lieberman, Raj Singh and Chris Goad. *W3C Geospatial Vocabulary*. `http://www.w3.org/2005/Incubator/geo/XGR-geo/`, October 2007. Accessed: 2015-09-05. (Cited on page 100.)

[López-de Armentia 2014] Juan López-de Armentia, Diego Casado-Mansilla and Diego López-de Ipiña. *Making social networks a means to save energy*. Journal of Network and Computer Applications, 2014. (Cited on page 39.)

[MacGillivray 2013] Carrie MacGillivray, Vernon Turner and Denise Lund. *Worldwide Internet of Things (IoT) 2013–2020 Forecast: Billions of Things, Trillions of Dollars*. IDC. Doc, vol. 243661, no. 3, 2013. (Cited on pages 2 and 64.)

[Mayer 2011] Simon Mayer and Dominique Guinard. *An extensible discovery service for smart things.* In Proceedings of the Second International Workshop on Web of Things, page 7. ACM, 2011. (Cited on pages 35 and 36.)

[Mayer 2012] Simon Mayer, Dominique Guinard and Vlad Trifa. *Searching in a web-based infrastructure for smart things.* In Internet of Things (IOT), 2012 3rd International Conference on the, pages 119–126. IEEE, 2012. (Cited on page 36.)

[Mayer 2014a] Simon Mayer. *Interacting with the Web of Things.* PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22203, 2014, 2014. (Cited on pages 2, 33, 34 and 36.)

[Mayer 2014b] Simon Mayer, Yassin N Hassan and Gábor Sörös. *A magic lens for revealing device interactions in smart environments.* In SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications, page 9. ACM, 2014. (Cited on pages 34 and 35.)

[Mayer 2014c] Simon Mayer, Nadine Inhelder, Ruben Verborgh, Rik Van de Walle and Friedemann Mattern. *Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning.* In Internet of Things (IOT), 2014 International Conference on the, pages 61–66. IEEE, 2014. (Cited on pages 2 and 38.)

[Mayer 2014d] Simon Mayer and Gabor Soros. *User Interface Beaming–Seamless Interaction with Smart Things Using Personal Wearable Computers.* In Wearable and Implantable Body Sensor Networks Workshops (BSN Workshops), 2014 11th International Conference on, pages 46–49. IEEE, 2014. (Cited on pages 33 and 34.)

[Montenegro 2007] G. Montenegro, N. Kushalnagar, J. Hui and D. Culler. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks.* RFC 4944 (Proposed Standard), September 2007. Updated by RFCs 6282, 6775. (Cited on page 1.)

[Nazzi 2011] Elena Nazzi and Tomas Sokoler. *Walky for embodied microblogging: sharing mundane activities through augmented everyday objects.* In Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services, pages 563–568. ACM, 2011. (Cited on page 39.)

[Nissenbaum 2009] Helen Nissenbaum. Privacy in context: Technology, policy, and the integrity of social life. Stanford University Press, 2009. (Cited on page 182.)

[Nitti 2014a] Michele Nitti, Luigi Atzori and Irena Pletikosa Cvijikj. *Network navigability in the social Internet of Things.* In Internet of Things (WF-IoT), 2014 IEEE World Forum on, pages 405–410. IEEE, 2014. (Cited on page 41.)

[Nitti 2014b] Michele Nitti, Roberto Girau and Luigi Atzori. *Trustworthiness management in the social Internet of things*. Knowledge and Data Engineering, IEEE Transactions on, vol. 26, no. 5, pages 1253–1266, 2014. (Cited on page 41.)

[Nottingham 2010] M. Nottingham and E. Hammer-Lahav. *Defining Well-Known Uniform Resource Identifiers (URIs)*. RFC 5785 (Proposed Standard), April 2010. (Cited on page 118.)

[Nottingham 2014] M. Nottingham. *URI Design and Ownership*. RFC 7320 (Best Current Practice), July 2014. (Cited on page 118.)

[Odersky 2004] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman and Matthias Zenger. *The Scala language specification*, 2004. (Cited on page 142.)

[Omicini 2004] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi and Luca Tummolini. *Coordination artifacts: Environment-based coordination for intelligent agents*. In Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1, pages 286–293. IEEE Computer Society, 2004. (Cited on page 48.)

[Omicini 2008] Andrea Omicini, Alessandro Ricci and Mirko Viroli. *Artifacts in the A&A meta-model for multi-agent systems*. Autonomous agents and multi-agent systems, vol. 17, no. 3, pages 432–456, 2008. (Cited on pages 47 and 154.)

[Ortiz 2014] Antonio M Ortiz, DH Ali, Soochang Park, Son N Han and Noel Crespi. *The cluster between Internet of Things and social networks: Review and research challenges*. 2014. (Cited on page 39.)

[Ossowski 2012] Sascha Ossowski. Agreement technologies, volume 8. Springer Science & Business Media, 2012. (Cited on pages 51 and 52.)

[Ostermaier 2010] Benedikt Ostermaier, K Romer, Friedemann Mattern, Michael Fahrmair and Wolfgang Kellerer. *A real-time search engine for the web of things*. In Internet of Things (IOT), 2010, pages 1–8. IEEE, 2010. (Cited on page 36.)

[Perez de Almeida 2013] Ricardo Aparecido Perez de Almeida, Michael Blackstock, Rodger Lea, Roberto Calderon, Antonio Francisco do Prado and Helio Crestana Guardia. *Thing broker: A twitter for things*. In Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication, pages 1545–1554. ACM, 2013. (Cited on page 39.)

[Pintus 2012] Antonio Pintus, Davide Carboni and Andrea Piras. *Paraimpu: a platform for a social web of things*. In Proceedings of the 21st international conference companion on World Wide Web, pages 401–404. ACM, 2012. (Cited on page 39.)

[Pinyol 2013] Isaac Pinyol and Jordi Sabater-Mir. *Computational trust and reputation models for open multi-agent systems: a review.* Artificial Intelligence Review, vol. 40, no. 1, pages 1–25, 2013. (Cited on page 52.)

[Platon 2007] Eric Platon, Marco Mamei, Nicolas Sabouret, Shinichi Honiden and H Van Dyke Parunak. *Mechanisms for environments in multi-agent systems: Survey and opportunities.* Autonomous Agents and Multi-Agent Systems, vol. 14, no. 1, pages 31–47, 2007. (Cited on page 47.)

[Poslad 2007] Stefan Poslad. *Specifying protocols for multi-agent systems interaction.* ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 2, no. 4, page 15, 2007. (Cited on page 50.)

[Prud'hommeaux 2014] Eric Prud'hommeaux and Gavin Carothers. *RDF 1.1 Turtle - Terse RDF Triple Language, W3C Recommendation 25 February 2014.* W3C Recommendation, World Wide Web Consortium (W3C), February 25 2014. (Cited on page 143.)

[Randall 2003] Dave Randall. *Living inside a smart home: A case study.* In Inside the smart home, pages 227–246. Springer, 2003. (Cited on pages 2 and 33.)

[Rao 1995] Anand S Rao, Michael P Georgeff*et al. BDI agents: From theory to practice.* In ICMAS, volume 95, pages 312–319, 1995. (Cited on page 46.)

[Rao 1996] Anand S Rao. *AgentSpeak (L): BDI agents speak out in a logical computable language.* In Agents Breaking Away, pages 42–55. Springer, 1996. (Cited on page 155.)

[Ricci 2007a] Alessandro Ricci, Mirko Viroli and Andrea Omicini. *CArtAgO: A framework for prototyping artifact-based environments in MAS.* In Environments for Multi-Agent Systems III, pages 67–86. Springer, 2007. (Cited on page 48.)

[Ricci 2007b] Alessandro Ricci, Mirko Viroli and Andrea Omicini. *Give agents their artifacts: the A&A approach for engineering working environments in MAS.* In Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems, page 150. ACM, 2007. (Cited on page 154.)

[Ricci 2009] Alessandro Ricci, Michele Piunti, Mirko Viroli and Andrea Omicini. *Environment programming in CArtAgO.* In Multi-Agent Programming:, pages 259–288. Springer, 2009. (Cited on pages 154 and 157.)

[Ricci 2011] Alessandro Ricci, Michele Piunti and Mirko Viroli. *Environment programming in multi-agent systems: an artifact-based perspective.* Autonomous Agents and Multi-Agent Systems, vol. 23, no. 2, pages 158–192, 2011. (Cited on page 47.)

[Rietzler 2013] Michael Rietzler, Julia Greim, Marcel Walch, Florian Schaub, Björn Wiedersheim and Michael Weber. *homeBLOX: introducing process-driven home automation*. In Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication, pages 801–808. ACM, 2013. (Cited on page 38.)

[Roduner 2007] Christof Roduner, Marc Langheinrich, Christian Floerkemeier and Beat Schwarzentrub. *Operating appliances with mobile phones–strengths and limits of a universal interaction device*. In Pervasive Computing, pages 198–215. Springer, 2007. (Cited on page 34.)

[Romer 2010] Kay Romer, Benedikt Ostermaier, Friedemann Mattern, Michael Fahrmair and Wolfgang Kellerer. *Real-time search for real-world entities: A survey*. Proceedings of the IEEE, vol. 98, no. 11, pages 1887–1902, 2010. (Cited on pages 35 and 36.)

[Rose 2014] David Rose. Enchanted objects: Design, human desire, and the internet of things. Simon and Schuster, 2014. (Cited on page 34.)

[Rukzio 2006a] Enrico Rukzio. *Physical mobile interactions: Mobile devices as pervasive mediators for interactions with the real world*. PhD thesis, University of Munich, 2006. (Cited on pages 33 and 34.)

[Rukzio 2006b] Enrico Rukzio, Karin Leichtenstern, Vic Callaghan, Paul Holleis, Albrecht Schmidt and Jeannette Chin. *An experimental comparison of physical mobile interaction techniques: Touching, pointing and scanning*. In UbiComp 2006: Ubiquitous Computing, pages 87–104. Springer, 2006. (Cited on page 34.)

[Sabater 2005] Jordi Sabater and Carles Sierra. *Review on computational trust and reputation models*. Artificial intelligence review, vol. 24, no. 1, pages 33–60, 2005. (Cited on page 52.)

[Sakimura 2014] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros and C. Mortimore. *OpenID Connect Core 1.0*. `http://www.openid.net/specs/openid-connect-core-1_0.html`, February 2014. Accessed: 2014-05-15. (Cited on pages 17 and 121.)

[Sambra 2015a] Andrei Sambra and Stéphane Corlosquet. *WebID 1.0 - Web Identity and Discovery, W3C Editor's Draft 30 April 2015*. W3C IG Editor's draft, World Wide Web Consortium (W3C), April 30 2015. (Cited on pages 121, 141, 142, 145, 147, 150 and 161.)

[Sambra 2015b] Andrei Sambra and Stephane Corlosquet. *WebID 1.0, Web Identity and Discovery*. `https://dvcs.w3.org/hg/WebID/raw-file/tip/spec/identity-respec.html`, 2015. Accessed: 2015-02-09. (Cited on pages 17 and 106.)

[Sauermann 2008] Leo Sauermann and Richard Cyganiak. *Cool URIs for the Semantic Web*. W3C Interest Group Note, 2008. (Cited on page 105.)

[Schmid 2007] Thomas Schmid and Mani B Srivastava. *Exploiting social networks for sensor data sharing with SenseShare*. Center for Embedded Network Sensing, 2007. (Cited on page 39.)

[Searle 1969] John R Searle. Speech acts: An essay in the philosophy of language. Cambridge university press, 1969. (Cited on page 48.)

[Shelby 2012] Z. Shelby, S. Chakrabarti, E. Nordmark and C. Bormann. *Neighbor Discovery Optimization for IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs)*. RFC 6775 (Proposed Standard), November 2012. (Cited on page 1.)

[Shelby 2014a] Z. Shelby, K. Hartke and C. Bormann. *The Constrained Application Protocol (CoAP)*. RFC 7252 (Proposed Standard), June 2014. (Cited on pages 1, 25, 101, 112, 114, 115 and 182.)

[Shelby 2014b] Zach Shelby, Klaus Hartke and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. 2014. (Cited on page 30.)

[Sherchan 2013] Wanita Sherchan, Surya Nepal and Cecile Paris. *A survey of trust in social networks*. ACM Computing Surveys (CSUR), vol. 45, no. 4, page 47, 2013. (Cited on page 52.)

[Shoham 1993] Yoav Shoham. *Agent-oriented programming*. Artificial intelligence, vol. 60, no. 1, pages 51–92, 1993. (Cited on page 47.)

[Sichman 1998] Jaime Simao Sichman, Yves Demazeau, Rosaria Conte and Cristiano Castelfranchi. *A social reasoning mechanism based on dependence networks*. In Proceedings of 11th European Conference on Artificial Intelligence, pages 416–420, 1998. (Cited on page 50.)

[Sichman 2001] J Sichman and Yves Demazeau. *On social reasoning in multi-agent systems*. Inteligencia Artificial, vol. 5, no. 13, 2001. (Cited on page 50.)

[Singh 1998] Munindar P Singh. *Agent communication languages: Rethinking the principles*. Computer, vol. 31, no. 12, pages 40–47, 1998. (Cited on page 49.)

[Snell 2014] J. Snell. *Prefer Header for HTTP*. RFC 7240 (Proposed Standard), June 2014. (Cited on page 146.)

[Speicher 2015] Steve Speicher, John Arwe and Ashok Malhotra. *Linked Data Platform 1.0, W3C Recommendation 26 February 2015*. W3C Recommendation, World Wide Web Consortium (W3C), February 26 2015. (Cited on pages 21, 22, 32, 42, 115, 145 and 182.)

[Sporny 2014] Manu Sporny, Greg Kellogg and Markus Lanthaler. *JSON-LD 1.0 - A JSON-based Serialization for Linked Data, W3C Recommendation 16 January 2014.* W3C Recommendation, World Wide Web Consortium (W3C), February 25 2014. (Cited on page 32.)

[Streitz 2005] Norbert A Streitz, Carsten Rocker, Thorsten Prante, Daniel van Alphen, Richard Stenzel and Carsten Magerkurth. *Designing smart artifacts for smart environments.* Computer, vol. 38, no. 3, pages 41–49, 2005. (Cited on page 34.)

[Sycara 1998] Katia P Sycara. *Multiagent systems.* AI magazine, vol. 19, no. 2, page 79, 1998. (Cited on page 46.)

[Takayama 2012] Leila Takayama, Caroline Pantofaru, David Robson, Bianca Soto and Michael Barry. *Making technology homey: finding sources of satisfaction and meaning in home automation.* In Proceedings of the 2012 ACM Conference on Ubiquitous Computing, pages 511–520. ACM, 2012. (Cited on pages 2 and 33.)

[Vaquero 2014] Luis M Vaquero and Luis Rodero-Merino. *Finding your way in the fog: Towards a comprehensive definition of fog computing.* ACM SIGCOMM Computer Communication Review, vol. 44, no. 5, pages 27–32, 2014. (Cited on page 82.)

[Vazquez 2008] Juan Ignacio Vazquez and Diego Lopez-De-Ipina. *Social devices: autonomous artifacts that communicate on the internet.* In The Internet of Things, pages 308–324. Springer, 2008. (Cited on page 40.)

[Verborgh 2011] Ruben Verborgh, Thomas Steiner, DV Deursen, Rik Van de Walle and J Gabarró Vallés. *Efficient runtime service discovery and consumption with hyperlinked RESTdesc.* In Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on, pages 373–379. IEEE, 2011. (Cited on page 38.)

[W3C OWL Working Group 2012] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation 11 December 2012.* W3C Recommendation, World Wide Web Consortium (W3C), December 11 2012. (Cited on pages 21 and 100.)

[Webber 2010] Jim Webber, Savas Parastatidis and Ian Robinson. Rest in practice: Hypermedia and systems architecture. " O'Reilly Media, Inc.", 2010. (Cited on pages 12, 114, 116 and 117.)

[Weiser 1991] Mark Weiser. *The computer for the 21st century.* Scientific american, vol. 265, no. 3, pages 94–104, 1991. (Cited on pages 1 and 33.)

[Weyns 2005] Danny Weyns, H Van Dyke Parunak, Fabien Michel, Tom Holvoet and Jacques Ferber. *Environments for multiagent systems state-of-the-art* *and research challenges.* In Environments for multi-agent systems, pages 1–47. Springer, 2005. (Cited on pages 46 and 47.)

[Weyns 2007] Danny Weyns, Andrea Omicini and James Odell. *Environment as a first class abstraction in multiagent systems.* Autonomous agents and multi-agent systems, vol. 14, no. 1, pages 5–30, 2007. (Cited on page 47.)

[Wilde 2007] Erik Wilde. *Putting things to REST.* School of Information, 2007. (Cited on pages 1, 23 and 37.)

[Wooldridge 1995] Michael Wooldridge and Nicholas R Jennings. *Intelligent agents: Theory and practice.* The knowledge engineering review, vol. 10, no. 02, pages 115–152, 1995. (Cited on page 46.)

[Yeung 2009] Ching-man Au Yeung, Ilaria Liccardi, Kanghao Lu, Oshani Seneviratne and Tim Berners-Lee. *Decentralization: The future of online social networking.* In W3C Workshop on the Future of Social Networking Position Papers, volume 2, pages 2–7, 2009. (Cited on page 16.)

[Zambonelli 2003] Franco Zambonelli, Nicholas R Jennings and Michael Wooldridge. *Developing multiagent systems: The Gaia methodology.* ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 12, no. 3, pages 317–370, 2003. (Cited on page 46.)

[Zhang 2012] Chunhong Zhang, Cheng Cheng and Yang Ji. *Architecture design for social web of things.* In Proceedings of the 1st International Workshop on Context Discovery and Data Mining, page 3. ACM, 2012. (Cited on page 39.)

École Nationale Supérieure des Mines
de Saint-Étienne

NNT : 2016 EMSE 0813

Andrei CIORTEA

WEAVING THE SOCIAL WEB OF THINGS : ENABLING AUTONOMOUS AND FLEXIBLE INTERACTION IN THE INTERNET OF THINGS

Speciality : COMPUTER SCIENCE

Keywords : Web of Things, Internet of Things, Multi-Agent Systems, Semantic Web

Abstract :

The Internet of Things (IoT) aims to create a global ubiquitous ecosystem composed of large numbers of heterogeneous devices. To achieve this vision, the World Wide Web is emerging as a suitable candidate to interconnect IoT devices and services at the application layer into a Web of Things (WoT).

However, the WoT is evolving towards large silos of things, and thus the vision of a global ubiquitous ecosystem is not fully achieved. Furthermore, even if the WoT facilitates mashing up heterogeneous IoT devices and services, existing approaches result in static IoT mashups that cannot adapt to dynamic environments and evolving user requirements. The latter emphasizes another well-recognized challenge in the IoT, that is enabling people to interact with a vast, evolving, and heterogeneous IoT.

To address the above limitations, we propose an architecture for an open and self-governed IoT ecosystem composed of people and things situated and interacting in a global environment sustained by heterogeneous platforms. Our approach is to endow things with autonomy and apply the social network metaphor to create flexible networks of people and autonomous things. We base our approach on results from multi-agent and WoT research, and we call the envisioned IoT ecosystem the Social Web of Things.

Our proposal emphasizes heterogeneity, discoverability and flexible interaction in the IoT. In the same time, it provides a low entry-barrier for developers and users via multiple layers of abstraction that enable them to effectively cope with the complexity of the overall ecosystem. We implement several application scenarios to demonstrate these features.

École Nationale Supérieure des Mines
de Saint-Étienne

NNT : 2016 EMSE 0813

Andrei CIORTEA

TISSER LE WEB SOCIAL DES OBJETS : PERMETTRE UNE INTERACTION AUTONOME ET FLEXIBLE DANS L'INTERNET DES OBJETS

Spécialité: Informatique

Mots clefs : Web des Objets, Internet des Objets, Systèmes Multi-Agent, Web Sémantique

Résumé :

L'Internet des Objets (IoT) vise à créer un eco-système global et ubiquitaire composé d'un grand nombre d'objets hétérogènes. Afin d'atteindre cette vision, le World Wide Web apparaît comme un candidat adapté pour interconnecter objets et services à la couche applicative en un Web des Objets (WoT).

Cependant l'évolution actuelle du WoT produit des silos d'objets et empêche ainsi la mise en place de cette vision. De plus, même si le Web facilite la composition d'objets et services hétérogènes, les approches existantes produisent des compositions statiques incapables de s'adapter à des environnements dynamiques et des exigences évolutives. Un autre défi est à relever: permettre aux personnes d'interagir avec le vaste, évolutif et hétérogène IoT.

Afin de répondre à ces limitations, nous proposons une architecture pour IoT ouvert et auto-gouverné, constitué de personnes et d'objets situés, en interaction avec un environnement global via des plateformes hétérogènes. Notre approche consiste de rendre les objets autonomes et d'appliquer la métaphore des réseaux sociaux afin de créer des réseaux flexibles de personnes et d'objets. Nous fondons notre approche sur les résultats issus des domaines des multi-agents et du WoT afin de produit un WoT Social.

Notre proposition prend en compte les besoins d'hétérogénéité, de découverte et d'interaction flexible dans l'IoT. Elle offre également un coût minimal pour les développeurs et les utilisateurs via différentes couches d'abstraction permettant de limité la complexité de cet éco-système. Nous démontrons ces caractéristiques par la mise en oeuvre de plus scénarios applicatifs.