Technical University of Munich

TUM Department of Electrical and Computer Engineering

Master Thesis

Runtime System Adaptation in Web of Things

Jan Lauinger

Assistant Professorship of Embedded Systems and Internet of Things
TUM Department of Electrical and Computer Engineering
Technical University of Munich

Runtime System Adaptation in Web of Things

Jan Lauinger

Master Thesis
Department of Electrical and Computer Engineering
Technical University of Munich

Supervisor	Prof. Dr. Sebastian Steinhorst Assistant Professorship of Embedded Systems and Internet of Things
Advisor	M. Sc. Ege Korkan
Author	Jan Lauinger
Date	September 27, 2018

Declaration of Ownership

I assure the single-handed composition of this thesis only supported by declared resources.

Munich, Date, Signature

Abstract

For building future industrial systems, the constantly emerging Internet of Things (IoT) requires interoperability between ambient devices. Individual solutions for IoT systems invite re-implementation of well known problems. They thereby slow system development in the IoT. The Web of Things (WoT) tries to achieve interoperability of IoT devices by leveraging well-functioning technology standards of the Web. The World Wide Web Consortium (W3C) proposes an WoT standardization concept which tackles the problem of re-development of already solved issues. Their standardization concept describes every IoT device with a Thing Description (TD). Based on the TD and associated functionalities, Web as well as IoT services are able to interoperate.

This work contributes to TD development by updating the TD testing tool Testbench to latest TD specifications. It moreover transforms the Testbench into a REST service before deploying it to the W3C Bundang plugfest 2018. The work proposes and implements a scalable and adaptive System Description (SD) for WoT mashups. The SD is compatible with the W3C WoT stack.

The main contribution of this work indicates that the SD of this work scales approximately linear while keeping verified runtime adaptation. By comparing WoT mashups with single or multiple SDs, it shows that mashups with multiple SDs scale better for larger system sizes.

With regard to the WoT research field, this work presents a TD extension with regard to system characteristics. This shows the expandability capabilities of the W3C WoT standardization. Furthermore, the implemented and evaluated SD represents an example to describe and manage WoT mashups in a W3C WoT compatible way. The developed SD mashup is an example for a scalable and adaptive WoT mashup implementation based on a SD concept.

Acknowledgements

I would like to offer my special thanks to Prof. Dr. Sebastian Steinhorst and all faculty members. This work would not have been possible without them. The project was challenging, educational, and motivational to me. I am particularly grateful for the assistance given by Ege Korkan who guided me through this project. His positivity, knowledge, and support inspired and motivated me to master all challenges. He was an exceptional advisor. Eventually, I would like to thank my family and close friends for the support and love.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADE	API Development Environment
API	Application Programming Interface
CoAP	Constrained Application Protocol
CoAPS	Secure Constrained Application Protocol
CPU	Central Processing Unit
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IoT	Internet of Things
IRE	Identifier, Resource, Entity
IRI	Internationalized Resource Identifier
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linking Data

LOV	Linked Open Vocabularies
MQTT	Message Queuing Telemetry Transport
NCS	Networked Control System
OD	Offering Description
OSI	Open Systems Interconnection
RAM	Random-access Memory
REST	Representational State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	The World Wide Web Consortium
WoT	Web of Things

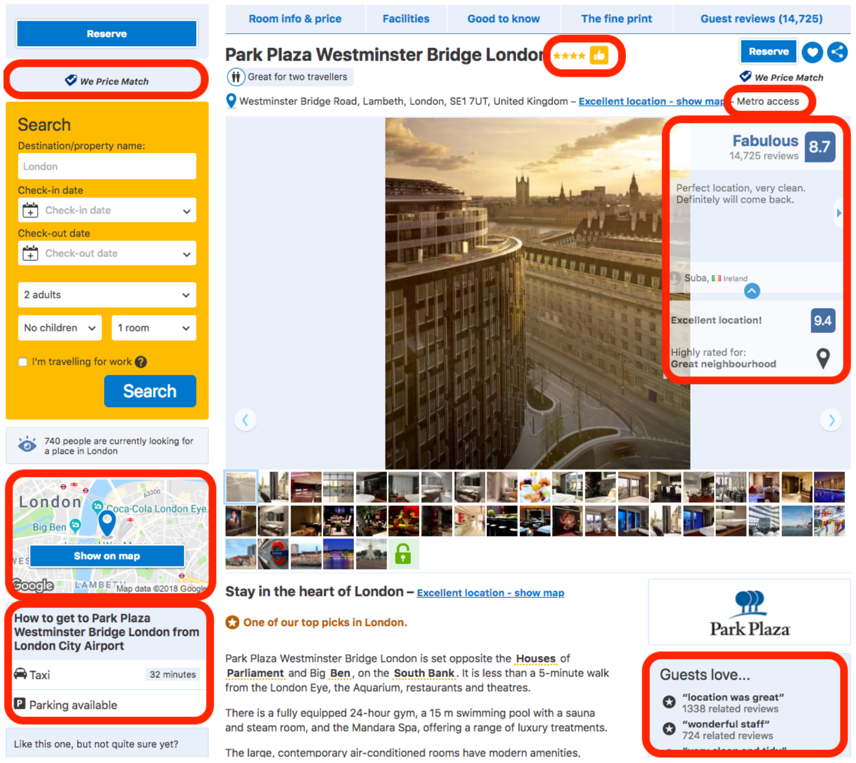
1. Introduction

This chapter motivates to solve interoperability problems in the Internet of Things (IoT) with the help of the Web of Things (WoT) concept. The WoT concept leverages standardized Web technology for solving interoperability issues in IoT systems. After an overview of existing WoT system solutions, this chapter presents research questions with regard to standardized WoT system implementations. After listing the contributions of this work, an overview of the organization of this work concludes this chapter.

1.1. Motivation & Problem Statement

The IoT term encompasses smart devices with the ability to communicate to the Internet. As a consequence, an IoT device provides remote controlling functionality and smartness. Multiple application fields of smart devices result in an opaque understanding of the IoT term in general. Smart industry, transport, home, and city solutions count as such application fields of IoT devices [10]. Products in these application fields often require systems with many connected devices with each providing different functionalities. IoT enabling technologies, protocols, and applications bring versatility and arbitrariness. Resulting complexity, management, and interoperability problems of IoT devices call for better connectivity, scalability, and standardization for IoT system solutions [11].

Secondly, there is the World Wide Web or simply Web which evolved from a document linking computer network to a network of connected services [12]. Today's websites consist of multiple services which contribute to the resulting user experience. Figure 1 shows service variety of the Booking.com website. The red marked boxes highlight information of additional services for the shown location. The price matching tag in the top left corner as well as user rating and recommendation information on the right half if Figure 1 mark possibilities to use internal or external application programming interface (API) services. An example for external service integration is Google Maps and a taxi service in the bottom left corner. Web



Park Plaza Westminster Bridge London ★★★★

Great for two travellers

Westminster Bridge Road, Lambeth, London, SE1 7UT, United Kingdom – Excellent location - show map - Metro access

Fabulous 8.7
14,725 reviews

Perfect location, very clean. Definitely will come back.

Suba, Ireland

Excellent location! 9.4

Highly rated for:
Great neighbourhood

740 people are currently looking for a place in London

How to get to Park Plaza Westminster Bridge London from London City Airport

Taxi 32 minutes

Parking available

Like this one, but not quite sure yet?

Stay in the heart of London – Excellent location - show map

One of our top picks in London.

Park Plaza Westminster Bridge London is set opposite the Houses of Parliament and Big Ben, on the South Bank. It is less than a 5-minute walk from the London Eye, the Aquarium, restaurants and theatres.

There is a fully equipped 24-hour gym, a 15 m swimming pool with a sauna and steam room, and the Mandara Spa, offering a range of luxury treatments.

The large, contemporary air-conditioned rooms have modern amenities,

Guests love...

"location was great"
1338 related reviews

"wonderful staff"
724 related reviews

Figure 1 Booking.com location search

services set successful examples for scalable, for the most part standardized, and manageable system implementations.

The Web of Things (WoT) concept tries to leverage successful methodologies of the Web to enhance IoT systems. It applies and adapts design patterns such as data formats, linked data, and media types of the Web to the IoT world. As a result, mechanisms such as resource discovery, creation, modification, and composition becomes available for IoT devices [13]. Concepts which made the Web scalable and successful should have the same effect in the WoT field and enhance WoT systems.

The World Wide Web Consortium (W3C) working group for WoT develops a standardization called Thing Description (TD) [14]. Thereafter, every internet compatible device or service counts as a Thing. Next to descriptive characteristics, a TD provides machine readability of Thing functionalities. Thereby, the abstract WoT TD concept allows to describe every IoT device or Web service. The standardization tackles and facilitates problems of machine to machine interaction of single devices. Development towards better Thing to Thing interoperation is important, as most WoT services aim to interconnect single devices into WoT systems [15]. WoT systems which consist of multiple devices or Things are also known as mashups.

Current system compositions of services and devices have arbitrary implementations due to different unique requirements [10]. It is always possible to come up with manual device composition solutions for WoT systems. Even tough current IoT application trends formulate recipes for system designs, there is no standardized way of describing a system [16]. This contributes to reasons why IoT device management solutions face reoccurring problems about flexibility, compatibility, security, and privacy. Web systems in comparison as shown in Figure 1 have the possibility to fall back on standardized communication protocols and architectural design principles. Additionally, the most used representational state transfer (REST) architectural style facilitates Web services to interoperate. The main assumption is that single Things as well as WoT systems align and benefit from existing successful architectural styles and standards of the Web.

The major success of Web standards allows the assumption of repetitive success with WoT concepts [17]. Not only the IoT field actively orientates itself by W3C standardization solutions of the Web [18]. Moreover automated and adaptive industrial systems search for potential solutions for problems that deal with scalability, security, interoperability, and simplified application integration solutions [19]. Even though state of the art WoT standardization focuses on single TD development, it is possible to specify a standardized description for WoT systems which builds upon TD approaches. Compared to single Things, WoT mashups face additional requirements implicitly.

To handle energy efficient, adaptive, secure, and scalable industrial systems, it is indispensable to leverage device and process monitoring system units [20]. Process monitoring and failure detection set the basis for system automation and adaptation. Especially adaptation requires the detection and awareness of system processes. Considerations about a standardized System Description (SD) have to take those requirements into account. Even though mashup applications are possible without TD compatible devices, a standardization approach to describe mashups would facilitate mashup adoption, applicability, and development [21].

Development of system determined solutions upon single Thing standards could lead to future goals of adaptive smart WoT solutions. Projects with strong economic impacts could benefit from the development of dynamic, and intelligently adaptive mashup systems. An example project is the smart farm with fine-scale sensing technology [22]. Their semantic sensor network with the capability to measure crop with the help of alerts strongly supports productivity and status awareness of farming goods. Farm-scale data opens opportunities and new benefits for industrial optimizations in services such as food processing, wholesaling, retailing, and consumption. Standardized SD standards with adaptive capabilities would further contribute to a satisfaction of facilitated implementation and scalability goals.

1.2. Existing Solutions & Research Questions

The W3C actively supports and hosts plugfests for TD standardization adoption and testing purposes. Participants of the TD standard which includes companies and other early adopters can test and contrast their own TD implementations within plugfests. Plugfests transfer existing developments of IoT mashup systems to the TD standard [23]. Plugfests enable the W3C to note problems with latest WoT standardization concepts. New findings and improvements support state of the art TD development. An example are TD developers which bring devices with TDs to plugfests. They often complain about Things that do not match their TD. This results in failing mashup compositions. The W3C WoT task force focuses on standardization issues and does not yet provide testing tools for TD validation. To further support adoption of TD standards, there is the need for tools which help to integrate, validate and test TDs on correctness. Moreover it is not specified how mashup developers interact with testing tools in a convenient and scalable way.

WoT devices with TDs have the goal of providing cross compatible systems. Such systems deal with automation and adaptation characteristics due to reliability reasons. Automated charging sets and example for adaptive control of device properties [24]. Automatic system adaption requires system state awareness and monitoring capabilities of components. This includes the ability to detect system failures and irregularities. A known issue is the fact that generated applications have a problem of recovering from component failures. Single feature failures might cause an orchestration of multiple Things to break. It is challenging to provide full failure detection inside such complex systems [16].

A SD could help to facilitate and anticipate complex problems regarding adaptive capabilities of mashups. TDs provide standardization for single components. They mark the first prerequisite for a standardized SD. Currently there is no SD which build upon existing standardized WoT solutions. Furthermore it is not clear if runtime adaptation works based on provided TD standards in system mashups. A SD could moreover solve larger scalability issues between systems and contribute to problems such as reliability and security. Basic requirements which allow automotive and adaptive system behavior have to be considered during the SD design process. It is thereby unclear if WoT methodologies such as semantics provide the possibility to solve the desired goals of the SD idea. By developing a SD based on existing standards, new needs for SD extensions and tools with regard to facilitation of runtime adaptation processes can be found. In the end it is possible to evaluate the design and solution of the SD on the specified criteria.

As a consequence of the stated problems, this work formulates the following research questions to with the aim to contribute and investigate findings for WoT runtime system adaptation:

- Is it valuable to extend and better integrate Thing Description testing tools for plugfests
- Which tool add-ons facilitate mashup developers the utilization of testing and validation tools
- Is it possible to provide scalable system runtime adaptation in Web of Things mashups
- Is it possible to define a System Description which uses the Web of Things stack
- Does a System Description contribute to scale runtime system adaptation of Web of Things mashups

1.3. Contribution

The first part of this work focused on TD based testing and validation. It was possible to and contribute effective extensions to the TD testing and validation tool called Testbench. The transformation of the Testbench software tool to a service allowed interactions through REST requests. A Testbench update ensured compatibility with the latest TD specification. Deployment of the updated and extended Testbench happened before the 2018 WoT Bundang plugfest. Resulting application results of the Testbench on TD testing and validation contributed to future WoT testing ideas and implementations.

The second part of this work proposed a SD for a WoT mashup with TD components. Afterwards, this work implemented the proposed SD which is compatible with the W3C WoT stack. This allowed WoT concept compatible Things to interact with the SD. The SD had the ability to expose new system specific features and characteristics to WoT mashups and environments. The SD had adaptive capabilities and was able to represent its characteristics based on system component availability. After SD mashup implementation, this work simulated failures of first single and then multiple Things and their capabilities. Failures found direct automated updates and changes within the SD of the system. When single SD updates worked, the same concept of failure adaptation found application for multiple Things and failures inside the system and its SD. The other main characteristic of the SD was its scalability. A SD evaluation setup observed better scalability of mashups with multiple SDs compared to mashups with single TD for larger mashup compositions.

In conclusion:

- This project improves the Thing Description Testbench:

1. Updates Testbench to meet latest Thing Description standards
2. Transforms Testbench into REST service

- This work proposes a WoT compatible System Description:

1. Modeling and description of System Description idea
2. Applies System Description to a WoT mashup with monitoring and adaptation capabilities

- This work evaluates the affect of the System Description on scalability and adaptivity of the mashup

1.4. Organization

This work starts by providing all necessary background information which the reader requires to understand the context of the research field. Afterwards the explanation of applied methodologies help to understand the SD proposal and Testbench update. After introducing the SD idea, this work divides the implementation chapter into two parts. The first part of the implementation describes updates of the Testbench. The second implementation part explains the implementation of the SD mashup and realization of the proposed SD idea. The evaluation chapter outlines SD behavior with regard to scalability and discusses flaws and implications. Next, the limitation chapter provides details of limitations of this work. The related works chapter connects and compares this work to other similar research findings and methodologies. In the end, the conclusion chapter summarizes the most important findings of this work and outlines future work.

2. Background

This chapter aims to explain main technical fields and components which contribute to the understanding of the project. The Web of Things (WoT) section explains fundamental concepts of the this work's main field. Afterwards, it introduces basic system concepts which will contribute to the WoT system design of this work.

2.1. The Web of Things

One approach to implement the WoT concept comes from the World Wide Web Consortium (W3C). It is necessary to understand fundamental components of the W3C standardization approach of WoT to be able to follow design decisions of the general system design of the project. In general, WoT connects two major fields. The first field is the Internet with the World Wide Web or simply Web. The Internet as a network of connected computers allows document or resource sharing of different data types. Standardized protocols thereby handle interoperability of diverse communication systems with the help of abstract layers. These layers are defined by the Open Systems Interconnection (OSI) model. This achievement of error-free communications across networks let the Web evolve from a simple read-only document sharing service to a global crowd and machine communication and interaction service. Latest Web development of semantic properties moreover have led to an increase of machine-readable content and hence smarter machine to machine interaction and decision making [25].

The other major field that contributes to the WoT term definition is the IoT. Embedded devices such as sensors, mobile phones and smart home appliances which are capable of communicating and sharing data over the Internet count as IoT devices. Most of these embedded systems come with limited but ever-increasing computing and communication capabilities. This brings new problems of limited packet size, packet loss, intermittent connectivity issues, and limited throughput, power, and complexity to those devices. Therefore, IoT research searches for better fitting solutions and adoptions of regular Web technologies for IoT devices [15].

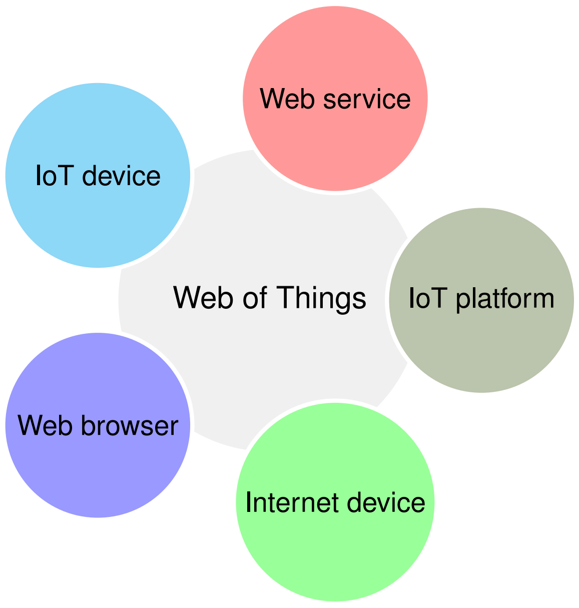


Figure 2 Web of Things as glue between the Web and the Internet of Things

The WoT term describes an approach to provide interoperability between physical IoT devices and Web technology. Figure 2 illustrates WoT as glue between different IoT, Web, and Internet services and devices. Researchers expected IoT devices to provide Web compatible concepts such as Representational State Transfer (REST) application programming interfaces (APIs) [26]. RESTful services enable device coupling through Uniform Resource Locators(URLs) as low level protocol semantics and standardized state exchange methods. Section 3.1 will explain the REST concept in detail. Current research in the IoT field applies popular Web technologies and mechanisms on physical devices [27] which marks application of WoT concept. Standardization which facilitates adoption of concepts helped the Web to evolve to the success and importance it has today. The WoT concept does not have standardized representation formats and link interactions types for building applications [28]. Most of today's IoT systems are manually scripted or composed with visual tools [29].

For establishing similar success as the Web, the W3C takes standardization development of WoT concepts into account. The institution which developed many of today's web standards and protocols follows specific patterns for creating a new standard in the WoT field. The W3C WoT working group has to fulfill all aspects defined by the WoT interest group in the WoT working group charter [14]. The scope of this charter lists four building blocks which compose the WoT protocol standardization. These building blocks are:

- Thing Description
- Scripting API
- Binding Templates
- Security and Privacy

During the standardization process, the W3C collects and updates new specifications within public working drafts. Information sources of this project rely on these public working drafts and not on more frequent changing editor's drafts.

W3C plugfests additionally contribute to standardization work. Plugfests allow developers and early adopters of WoT standards to test, integrate, and exchange insights about implementation details. The W3C further improves their WoT standardization building blocks with this information. Next subsections will describe these building blocks. But before that, another subsection about the WoT Architecture sets building blocks in context to each other and provides an overview over all W3C WoT specifications.

2.1.1. WoT Architecture

The WoT Architecture specifies the environment for TDs to interoperate across IoT Platforms and application domains. It therefore names general communication interfaces and implementation layers within Internet components. Figure 3 shows all architectural components and mentions interplay types. It thereby shows fundamental aspects for mechanisms with independent hardware to communicate with each other with maximum flexibility, compatibility, security, and privacy. The environment of Figure 3 divides itself into four parts. Architectural components live in cloud, gateway, local and Web environments. Every environment shown in the architecture has a WoT Servient component. Figure 4 shows the Servient component in greater detail but will be introduced after the architecture.

The architecture of Figure 3 uses blue arrows to indicate interaction possibilities between Servients. It differentiates communication purposes with typical examples. Hence, the arrow between the cloud and the gateway Servient showcases remote access and synchronization. Another arrow with an orchestration tag connects the gateway Servient to local Servients. Local devices show direct Servient to Servient interactions. Complement devices are able to communicate due to that fact that every device exposes a TD. What a TD exactly is will be explained in the next section of this chapter. For now, it is enough to know that TDs provide descriptive information for communication between Things. Every hardware device with Web compatibility counts as a Thing. Lastly, Figure 3 indicates that Web browsers are extendable towards

W3C WoT compatibility.

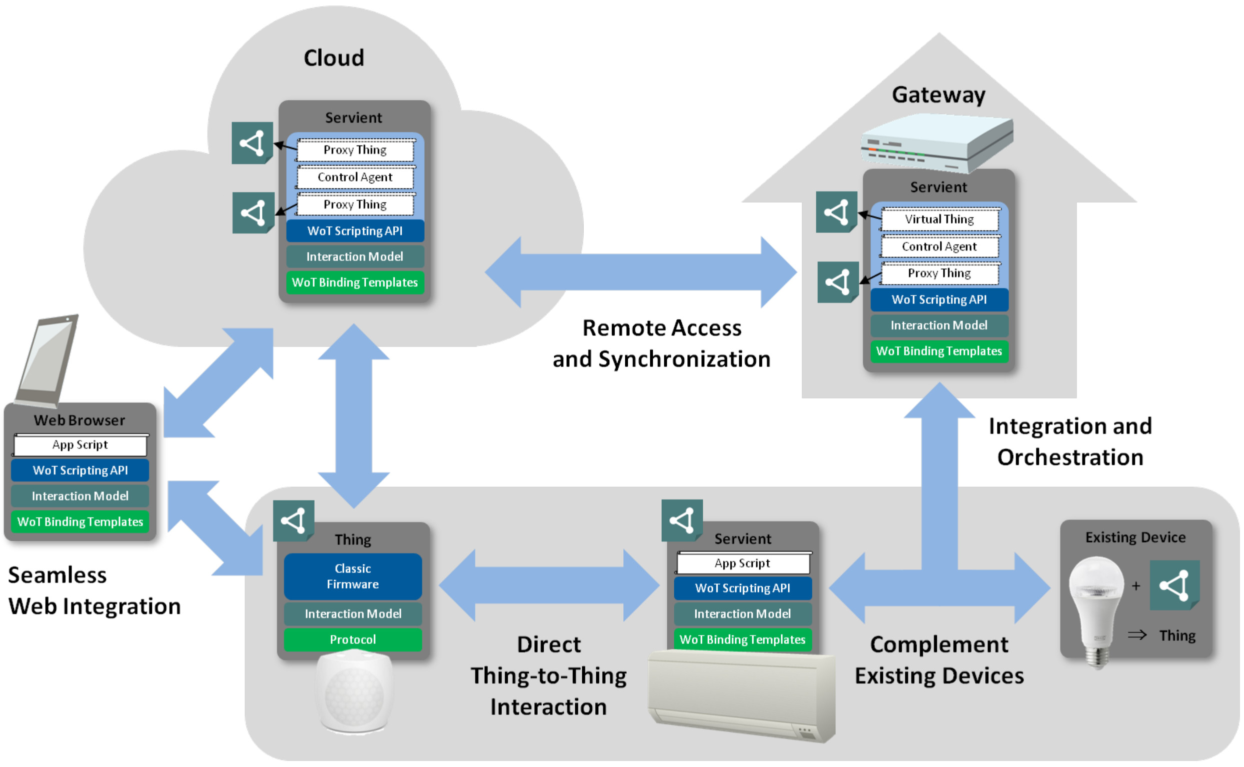


Figure 3 Abstract Architecture of W3C Web of Things specifications [1]

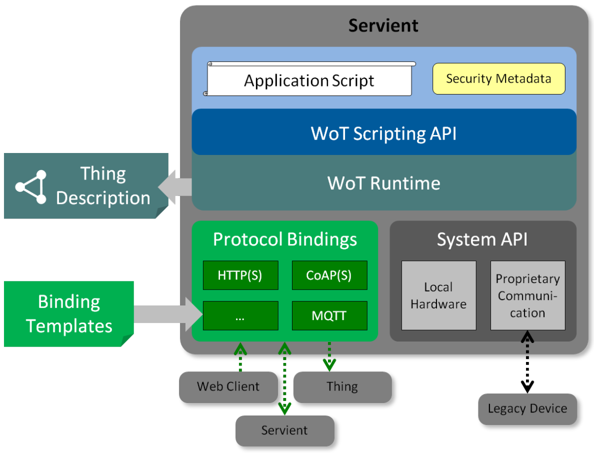


Figure 4 Component model of a Web of Things Servient [1]

WoT compatible Web browsers have limited protocol bindings due to the fact that the HTTP protocol is the standardized application layer protocol the Web. Especially the compatibility of the Servient architecture and web browsers makes WoT standardization applicable to the majority of web devices.

A Servient as in Figure 4 implements all WoT building blocks. This software stack uses the WoT runtime to expose the TD of the underlying device. On top of the runtime sits the WoT scripting API. The scripting API provides the possibility to submit application scripts with security data to the WoT runtime. Other components of the Servient of Figure 4 represent protocol bindings and a system API. With the system API, the W3C specifies the missing standardized link which allows to connect IoT hardware to the web. The protocol bindings of a Servient represent communication protocol compatibility. Similar to the introduction of the scripting API, details about protocol bindings are in upcoming subsections. Devices which implement W3C WoT Servients have the ability to recognize, discover, and communicate with other Things independently of their location. It is necessary to go into further detail of WoT Servient components to understand final system design criteria.

2.1.2. WoT Thing Description

The WoT TD can be seen as the most challenging and most important standardization part. This is due to the fact that it summarizes all components of the W3C Servient 4 in one description. A TD extends the formal interaction model of a device and has the ability to describe required device functionalities in a standardized description. For the ability to express all different kinds of information in a linked and structured way, TDs rely on the JSON-LD [30] data format which again heavily relies on the JSON data format. Even though JSON-LD manages to represent complex and dependent information, it stays humanly readable and writable. A remarkable feature of the underlying JSON format is that it helps web services to scale with its ideal and frequent applicability within unstructured databases. The standardized description takes all architectural W3C building blocks into account while keeping a modular structure which allows additions of specific on demand add-ons.

Listing 1 illustrates a TD. Line number 2 of the TD of Listing 1 shows general metadata of a TD. This information represents first semantic annotations of this TD. Semantic annotations as a feature of the JSON-LD data format links this TD to an Internationalized Resource Identifier (IRI) [31]. Such a capability JSON keys disambiguates and references JSON documents to a context. The underlying JSON format of the TD is based on key value pairs. Listing 1 marks keys in blue and values in black color. Semantics as well as the JSON-LD data format will be introduced in more detail in the next chapter 3. Semantic annotations fit into the category of additional on demand information of a TD. The following `id` and `name` keys count as general metadata as well.

Listing 1: JSON-LD 1.1 Thing Description sample with semantic extensions [2]

```
1  {
2      "@context": "https://w3c.github.io/wot/w3c-wot-td-context.jsonld",
3      "id": "urn:dev:wot:com:example:servient:lamp",
4      "name": "MyLampThing",
5      "properties": {
6          "status": {
7              "writable": false,
8              "observable": false,
9              "type": "string",
10             "forms": [{
11                 "href": "coaps://mylamp.example.com:5683/status",
12                 "mediaType": "application/json"
13             }]
14         }
15     },
16     "actions": {
17         "toggle": {
18             "forms": [{
19                 "href": "coaps://mylamp.example.com:5683/toggle",
20                 "mediaType": "application/json"
21             }]
22         }
23     },
24     "events": {
25         "overheating": {
26             "type": "string",
27             "forms": [{
28                 "href": "coaps://mylamp.example.com:5683/oh",
29                 "mediaType": "application/json"
30             }]
31         }},
32     "links": []
33 }
```

Line 5, 16, and 24 of Listing 1 indicate TD interaction types. Properties, actions, and events JSON keys gather core vocabulary definitions of a TD in lists. The core vocabulary of a TD consists of property, action and event. Properties, actions, and event JSON structures describe the interaction model of the Thing. These interactions affect the application functionality of a Thing heavily. The reason for this is that interaction links represent the method to expose device functionality to other Things. Each interaction field can contain multiple entries. The TD of Listing 1 contains a single property called `status` inside its `properties` key. Line 6, 17, and 25 of Listing 1 mark interaction entries.

Every interaction type has own specification which differentiates itself of other interaction types. A property inside the `properties` JSON key can have `writable` and `observable` field names. These definitions specify whether a client is able to write or read the property. When clients interact with a Servients based on TD information, they can invoke actions. An interaction of type action typically causes the device to perform a procedure which might take time. Actions therefore tend to send a delayed response to the client. Properties of a TD usually expose static properties of the device which have no response delay. A client of a W3C is also capable of subscribing to events. If the device of a TD triggers an event, the client receives a notification.

TDs use JSON-Schema [32]. As well as other data structures such as JSON-LD, JSON-Schema will be introduced in Chapter 3. For now, it is sufficient to know that JSON-Schema indicators specify JSON structure. The JSON `type` key in line 9 and 26 of Listing 1 represents an JSON-Schema indicator. This indicator specifies the value type of the interaction entry. In the case of the `status` property of line 6 of Listing 1, the JSON-Schema indicator ensures that the value of this property is of type string. Inside the interaction JSON structures, the WoT protocol binding templates specify communication protocols of interactions [33]. The `forms` JSON key of the TD example of Listing 1 illustrates the use of the CoAP communication protocol. Details of protocol binding templates are in the WoT protocol binding templates subsection 2.1.5. Communication metadata derives from protocol binding templates and contributes to the TDs with data such as forms, interaction types, and data encoding. Forms and data encoding structures of the payload such as the `mediaType` key of Figure 1 represent communication metadata. Lastly, the `links` JSON key typically holds URLs to other TDs or devices in an array.

For a total view of possible TD features, there is the upcoming TD data model in Figure 5. This Figure shows all hierarchical connections between all TD attributes. In the top left corner of Figure 5, there is the Thing class which sets the base of the whole TD. All other classes of property, action, event, link, and form have a zero-to-many relationship between themselves and the Thing entity. This means that a Thing can have at least no or at maximum infinitely many attributes of those other class objects. Inside the Thing, attributes which are also TD JSON keys have a one-to-one and zero-to-one cardinality. A one-to-one mapping specifies a mandatory JSON key.

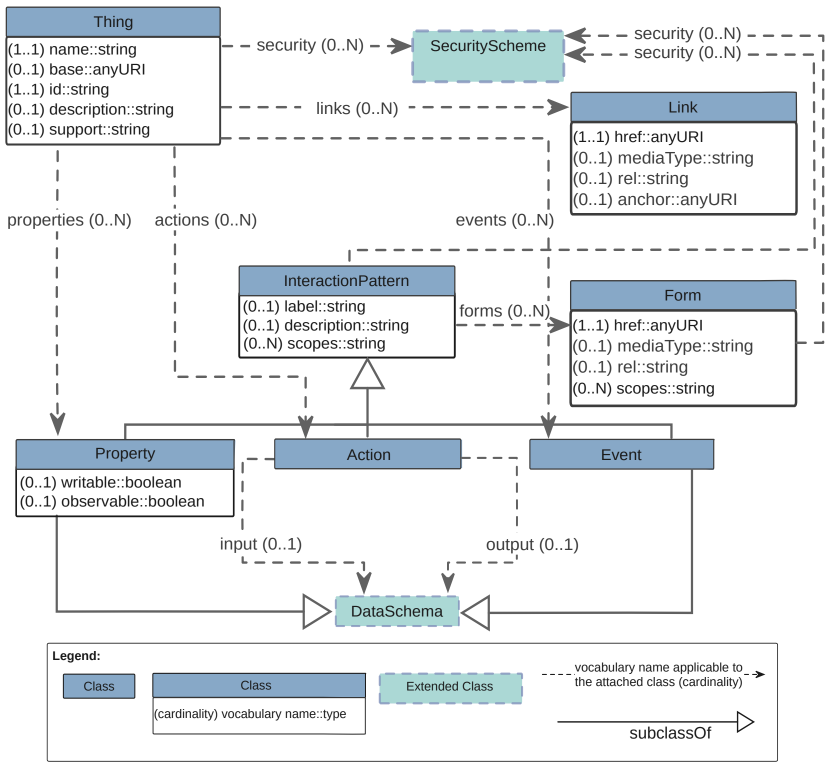


Figure 5 Thing Description core entity relation data model [2]

Explanatory paragraphs about Listing 1 already stated typical interaction use cases. Nevertheless, the description of Figure 5 reveals all possible functionalities of the TD entity relation schema in detail. Form the entity relation schema perspective, the property class equals the event class. A property has a writable as well as observable attributes. Setting writable to `false` as indicated by Listing 1 forbids a client to modify the value of the property object of the Thing. The observable attribute allows TD consumers to subscribe to a property or event if set to true. The terms of TD exposure or consumption derive from W3C scripting API specifications and will be introduced in Subsection 2.1.4. The general meaning of these terms indicate that TDs can be consumed or exposed. Subscription on a property implies notifications on status changes or event occurrences. Actions can have input and output attributes for data reception and sending. Interactions such as property, event, and the input and output fields of actions are subclasses of a data schema class.

The `dataSchema` field composes JSON-Schema and additional individual on demand keys. Links have the purpose to link whole TDs with other TDs. Hence, a consumer of a TD can automatically fetch these linked TD resources. The form class keeps communication protocol methods and options as well as payload media types together. An action is a subclass of an interaction pattern which relates to the form and security scheme classes. Interaction patterns provide additional attributes to non standardized action interactions. Arbitrarily chosen input and output attributes of actions profit from those implicitly inherited attributes of the form and security class to clearly specify the desired interactions.

The security scheme data model will be introduced in the WoT security and privacy subsection 2.1.6. Aside from that it is not necessary to explain the data model of the `dataSchema` class because of heavy resemblance with JSON-Schema specifications. Another important note is that the specifications of the W3C TD standardization will look different in soon future. This is due to the fact that WoT standardization working drafts receive updates during the time of writing this thesis. The next sections take the introduced TD vocabulary into account and specify further functionalities.

2.1.3. Interaction between Things

With TD information of the previous subsection, it is clear that TDs enable interactions between Things. Figure 6 showcases a typical interaction procedure that is possible with TD usage.

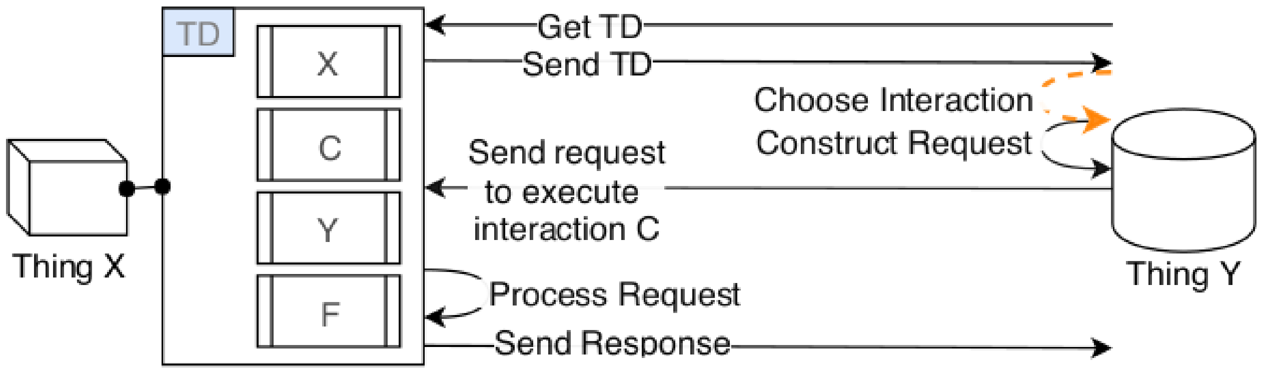


Figure 6 Interaction procedure between two Things [3]

On the left side of Figure 6, Thing X exposes its TD which contains different interactions marked as X,C,Y, and F. Thing Y on the right side of the graph first requests the TD of Thing X. After the consumption of the TD of Thing X, Thing Y has the possibility of choosing an interaction of Thing X. Calling an interaction at Thing Y causes a request to Thing X. This request executes the interaction exposed by Thing X. After the process of the executed interaction finishes, Thing X sends back a response to Thing Y.

This interaction example of two Things illustrates the interoperability of ambient devices that are capable to implement the W3C Thing definition. The interplay of two Things allows different interoperation purposes.

The next subsection provides a larger Thing to Thing communication scenario and explains all W3C WoT scripting API terms.

2.1.4. WoT Scripting API

With the understanding of the introduced underlying features of a Thing, it is possible to understand scripting API specifications. Scripting API methods allow execution of scripted commands on Things. Commands can implement a TD consumption, interaction calls, as well as exposure of own Thing features. The scripting API leverages actions to execute remote procedures with or without input and output data, property capabilities for reading and writing data, and events for subscriptions and notification reception. Figure 7 demonstrates scripting API functionalities in the context of two Things, two TD repositories and a proxy server as a Thing in the middle. It is not necessary to explain the type of proxy server since Figure 7 has the purpose of illustrating Thing to Thing interoperation with regard to scripting API terms. The task of a TD repository is to maintain and distribute registered TDs to other Things. With the aid of Figure 7, it is possible to see the discover, register, unregister, fetch, expose and consume methods of the scripting API that are proposed by the W3C.

The interplay scenario of Things of Figure 7 starts with a synchronization process. Before launching the synchronization, the first Thing exposes itself. The expose method of the scripting API allows to specify properties, actions, events, and moreover handler methods for every defined interaction. The top left corner of Figure 7 shows the registration of the TD of a device at the first TD repository. The initial registration enables other Things to discover the first TD at the TD repository. Awareness between Things is mandatory for discovery of TDs. Figure 7 illustrates that the proxy server Thing is able to discover the first Thing over the TD repository. The discovery method of the scripting API directly calls the consume method on the discovered TD. It is also possible to fetch a TD via an uniform resource identifier (URI) and explicitly consume it afterwards by calling the WoT object consume method. Opposed to the expose method, the consume method returns an object which enables to activate features of the consumed TD. Methods of the consume return object can utilize discussed interaction patterns

The input for the discovery an fetch methods only rely on a URI to a TD and do not depend on a TD repository. The TD repository Thing has the purpose of attracting TD discovery calls by gathering TDs. This design bundles discovery traffic and keeps it away of Things which can then respond to interaction calls only. The proxy server Thing of Figure 7 may expose and register its own TD at a second TD repository. Again another second Thing has the ability to discover the servers TD from the second TD repository. The second Thing can now consume the TD of the first Thing with the consumption of the server TD. Synchronization of these three Things ends with the perception of all TDs by the second device.

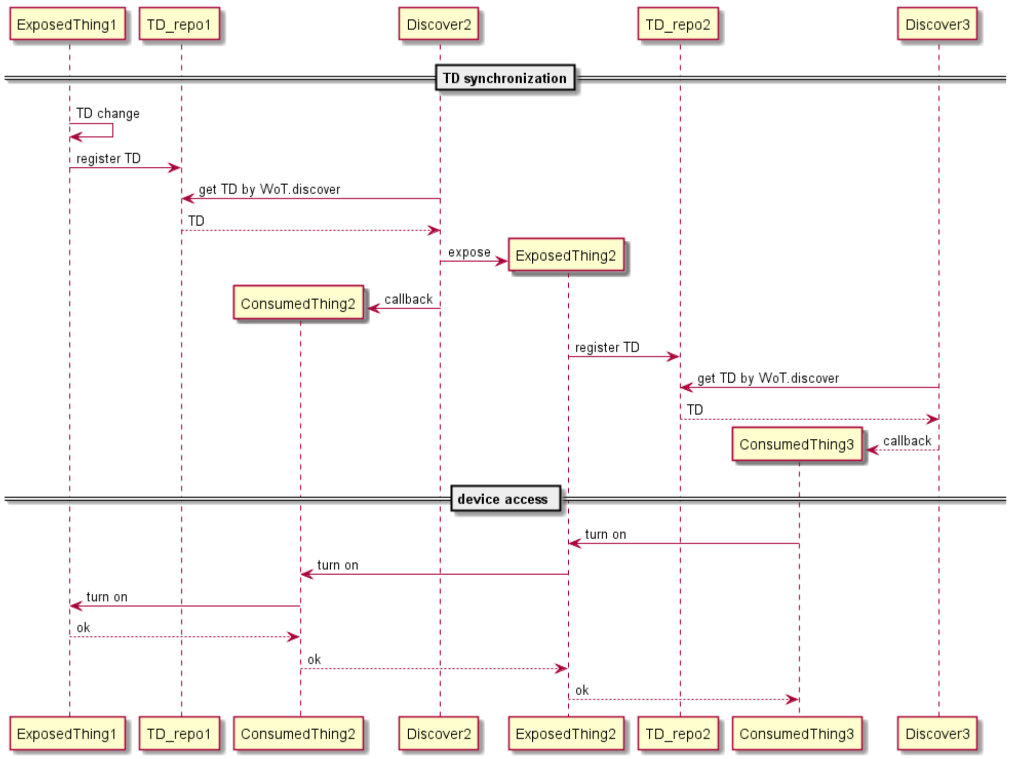


Figure 7 Flow graph of remote device to device access using the Web of Things scripting API [4]

After the synchronization, the second device is able to execute remote procedures by invoking actions or read and write properties on the first Thing. The other way around, the first Thing cannot call interactions of the second Thing because it does not consume anything. Device access is rather straightforward after synchronization of different Things completes. When the second device requests an interaction of property, action, or event on the first device, the request arrives first at the server. The server passes the request to the consumed first device. When the interaction of the first device succeeds, it responds back to the server which passes the responds back to the requesting second device.

2.1.5. WoT Binding Templates

A necessity of an interaction is the awareness of shared communication specifications. WoT binding templates solve this problem by defining the part of a TD which describes interaction design patterns [33].

Regarding data schema, the object payload structure is specified by binding templates information. Moreover WoT binding templates provide a standardized way to set value constrains. WoT binding templates take the form JSON keys to specify relation types, media types, protocols, and protocol options and methods.

Relation types differentiate expected results inside a form. Read and write interactions of a property for example can have different protocol forms and the relation type can indicate whether a form describes a read or write. Media types formulate the payload data type. Common used types are "application/json" or "plain/text". Protocols and protocol methods and options depend for the most part on form vocabulary such as HTTP, CoAP, MQTT, or the websocket protocol [34]. Bindings for property, action, and event vary typically in a way that CoAP observe or HTTP long polling finds more applicability to events instead of properties. HTTP PUT and GET requests serve property read and write interactions.

2.1.6. WoT Security and Privacy

The WoT security and privacy considerations outline security objectives and likely threads. But most importantly these guidelines ensure and preserve existing security mechanisms and functionalities inside TDs. Risks may occur during exposure, consumption, or communication with protocols. Therefore, the WoT security and privacy considerations recommend and provide example configurations for TD application scenarios such as home, corporate, and industrial environments [35]. Since the security and privacy considerations affect all WoT standardization fields and building blocks, TDs contain a security JSON key as a top level element which enables to specify different security configurations. The value to this security key is an array which contains security schemata. It is possible to implement basic user password authorization or OAuth 2.0 authentication [36] inside security schema subclasses to secure Thing to API or Thing to Thing communication.

2.2. WoT System Design

This chapter describes mashups with regard to state of the art WoT TD developments and implementations. Since this work proposes and implements a WoT SD, it is important to understand basic design principles and implication of WoT mashups. In a similar way as a TD describes a single Thing, a SD describes a mashup of Things. A a result, topics introduces in this section contributed to the SD design proposal of Chapter 4.

2.2.1. Mashups

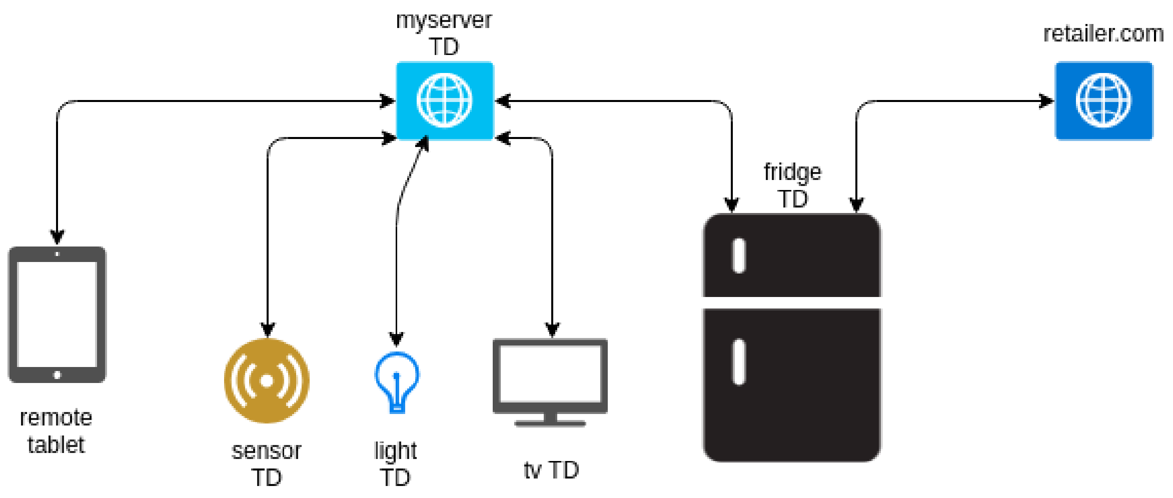


Figure 8 Web of Things smart home mashup

A mashup is a composition of devices or services which forms a new application. WoT mashups derive from Web infrastructures which leverage multiple Web services. It is common that Web services retrieve external documents with Web API calls. They collect and bring together information from different sources for producing a new application. An example is the website of the first chapter of Figure 1. A WoT mashup opens new opportunities. In WoT, IoT devices are Web compatible and allow to build mashups and applications that use physical values of sensors.

Figure 8 shows a potential WoT smart home mashup. Physical devices with TDs allow to communicate with Things in the Web. As Figure 8 shows, it is possible to use a remote but Web connected tablet for accessing a Thing in the Web. The web server as a Thing is able to communicate with home devices and expose their interactions or values. At the same time, Figure 8 indicates that Things can independently connect to other services for accessing diverse services. The composition of heterogeneous devices connected to a user interface marks a typical WoT example [37]. Devices have differentiating functionalities, capabilities, and producers. WoT mashups bring together diverse Things to enable new applications.

Generally, there are two types of WoT mashups. Those types are physical-virtual mashups and physical-physical mashups [15]. As the names suggest, physical-physical mashups consist of Web services from smart devices only. Physical-virtual mashups on the other hand consist of physical and virtually Web compatible services. For instance, a cloud hosted Web service represents a virtual Web service. The combination of physical and virtual Web services allows physical-virtual mashups to provide valuable user interfaces. Utilization of Web based user interfaces for monitoring physically collected sensor data marks an example case [21].

Since WoT mashups consist of multiple components, different interactions between single components are possible. Single components can affect overall system properties as well as subsystems. Single state of the art TDs implement properties such as temperature values [38], position sensor values [39], and alert indicators [40]. System compositions of TDs with these interactions could make up mashups with different interaction effects.

A first case of mashup interactions happens between Things of the system which consequently cause a system interaction. Examples for this first case set the following scenarios. The position of a car could cause an interior light to turn on. Multiple repetitions of this interactions could oblige a system interaction to emit an event. Similarly a sensor network allows to gather multiple values for writing to a system property. A second case of mashup interactions is that a system interaction provokes single Thing interactions inside the system. An example scenario for this case is a system interaction which causes Thing to relocate to the desired input. Lastly there is the case of mixture or dependent mashup interactions. This means that a system interaction affects a Thing interaction which again causes a system interaction. It is helpful to be aware of different mashup interaction types for designing the structure of a mashup.

2.2.2. System Components

The versatility of application scenarios for WoT Servients invites to think of complex mashup applications with sophisticated system requirements. Modern storage systems for instance use different components for controlling, connection, intelligence, and other purposes. Algorithms and protocols help system components to handle scalability, reliability, failure detection, and automated adaptation to failures [41]. Nevertheless the system components of a WoT mashup depend on two abstract component types only. This is due to the fact that WoT mashups may contain a TD, a SD, or both.

It is not specified whether a SD component follows a distributed or central implementation approach. The only declaration is that the SD component serves as a connection or management component. It is moreover not possible to specify how smart and computationally powerful the SD component should be. Properties of the connective SD component depend on mashup application requirements. Since mashup requirements of this work affect scalability and adaptation of the system, the SD component must allow TD orchestration through algorithms and protocols. Again there are multiple application scenarios which specify if it is best to implement a system component that contains a SD only. Or, if the system component works better in combination with a TD.

2.2.3. Runtime Adaptation

Before explaining runtime adaptation in the Web and IoT field, it is helpful to explain the basic definition. A runtime of a system is adaptable if it is able to react on events, system changes, failures, etc. Most often, a runtime requires one component for detecting a source which causes the adaptation procedure.

Runtime adaptation already exists in IoT as well as Web systems [42]. A necessary requirement for runtime adaptation is an always on system or process. This runtime process requires reliable uptime. Moreover, a running process must have the capability of monitoring states, failures, or other events happening inside the system. An example of an adaptive IoT runtime could be a wireless sensor network and an analyzer tool which is able to monitor the network state. A programmable analyzer tool could detect system irregularities such as failures for example. As a consequence, programmed code could react and adapt system components and properties with regard to the failure causing events.

Runtime adaptation of Web services often use real time data of user website interaction. Web services store user data to feed predictive models. Predictive models help to adapt content in multiple ways. Based on user interaction with a website, it is possible to place personalized messages or advertisement for Internet users [43]. Here, the always on component is the Web service. Storage of user interaction data enables monitoring and resulting adaptation opportunities.

WoT standardization concepts deal with single TDs for now. Development with the help of mashups claim for system compatibility of Things. As adaptive system in the IoT and Web world are common, WoT mashups can use adaptation concepts of both domains. This is an important fact which outlines that WoT mashup should not be restricted to certain adaptive measures. The only requirement for a WoT mashup is to provide compatibility with always on processes. Independent of modular or embedded controlling units, WoT mashups need compatibility for communication with controlling services. For example when a WoT SD expresses the current system state, it should be reachable for interval or constantly requesting controller services. This important design criteria enables runtime adaptation with the help of SDs. Another important note is that SDs have to have a dynamic data format which allows modifications. If a runtime sets a system property value during system commissioning, is must be able to overwrite or change this value during runtime. Otherwise adaptiveness cannot express itself.

Systems from the IoT and WoT world with monitoring prerequisites can be set up either semi automated or fully automated [44]. The semi automated approach requires a process designer compose executable service specifications. Fully automated system compositions compose, configure, and connect components without manual intervention. Fully automated mashups facilitate system recovery processes and support reliability of systems. Especially industrial systems require such capabilities due to reliability needs [45]. Therefore consideration about WoT SDs have to include multiple system designs. Since a SD contains system information, it represents a central component for contributing to adaptation capabilities of WoT mashups.

2.2.4. Scalability

Scalability of a system defines how a system behaves during growth. It is an important factor in upcoming WoT mashups. The reason for this is that industrial IoT system already use a high number of devices and services. Intelligent transportation systems mark an example [46]. Design of WoT systems plays an important role with regard to its scalability. Bad system design can lead to traffic congestion and system bottlenecks of availability and responsiveness. Thereby, the approach of central or distributed system component design is one of the first question to answer. Next to system design, scalability testing tools provide a way to ensure system scalability [47]. It is helpful to think of scalability simulation techniques in advance of designing a SD for WoT mashups.

2.3. Distributed vs Centralized Systems

Since bad system design can affect system performance, responsiveness, reliability, scalability and other system properties, it is important to introduce common approaches of system designs. System design can vary between three approaches. Those approaches are centralized, decentralized, and distributed. The decentralized approach lies in between the other approaches. This section explains system design principles with regard to a WoT SD.

In centralized computer systems, calculations happen on one specific component. Often this component is a dedicated server which gathers and calculates data of other system components. After the calculation, results become redistributed to other system components. Computationally constrained IoT sensor devices often send physical data to central management services for monitoring and analysis purposes. Due to computational constrains, IoT devices cannot process, control, and manage network data. The central approach therefore uses computational powerful devices on particular components where the computation is vital [11]. A big advantage of centrality is the facilitation of information management. Central information safekeeping allows fast data processing. Nevertheless scalability sets a big issue. Central computers cannot scale horizontally as distributed computing systems. Therefore, data accumulation can take up a lot of time in large computing systems. And redistribution of gained processing insight prolongs client service waiting times [48].

Distributed computer systems spread calculations to all computers inside the system. A typical use case is large scale data processing through distribution of big data sets to multiple computers [49]. Distributed computing allows parallel computations. Calculation results find further processing after distributed computers return processed data. Distributed systems mark important components in today's large scale Web services which utilize cluster computing processes [49]. Even though distributed systems solve problems of scalability, reliability, and system failure adaptation, every participating component of a system requires enough computing power [41]. This is due to the fact that every device in the system requires service processing capabilities to handle complex protocols. Distributed computing computations often shuffle redundant data [48] due to design patterns and complex protocols. This causes communication overhead and higher processing demands. As a result, a common requirement in distributed computing system is the availability of sufficient computing power.

Regarding WoT concepts, there are several important consideration to mention about central or distributed system design for WoT mashups. It is important to understand pros and cons of distributed and central computer systems with regard to SD distribution inside the system. Figure 9 shows centralized, decentralized, and distributed arrangements of network control systems (NCSs). All three NCS controllers control sensors and actuators which interact with a plant. The example illustrates communication, computation, and control of a monitoring system. WoT mashup components deal with the same requirements and need one controlling element for system adaptation purposes. Similarly to IoT concepts and the location of controlling units, WoT system architecture decisions have to consider the location of the SD. Gateways usually come with higher computing capabilities compared to sensor end devices. Therefore, gateway components frequently fulfill system management tasks. Since gateway components already set a candidate for performing management tasks, they represent a candidate for hosting a SD. The reason for this is

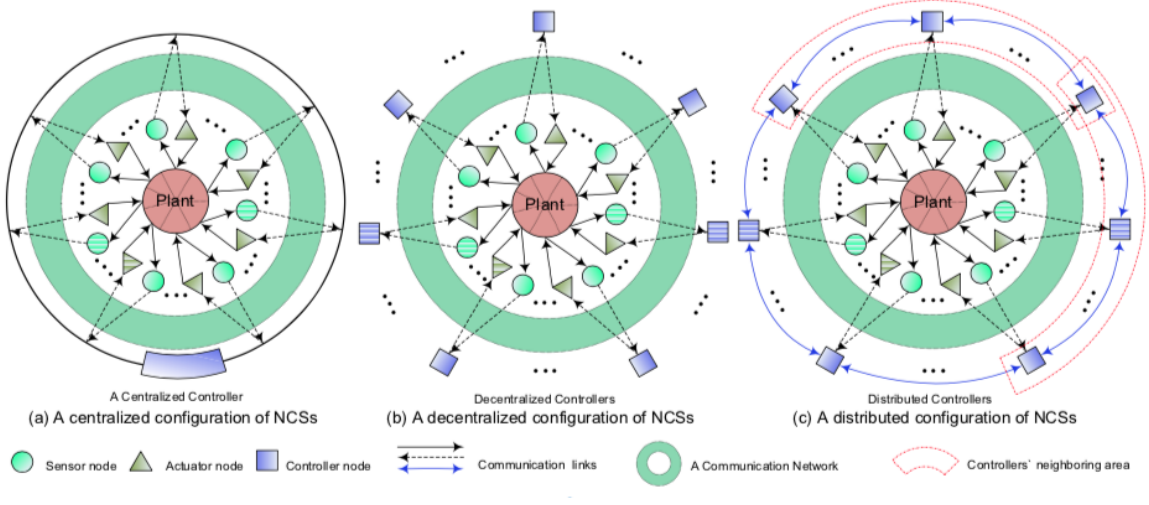

Plant
A Centralized Controller
(a) A centralized configuration of NCSs
Decentralized Controllers
(b) A decentralized configuration of NCSs
Distributed Controllers
(c) A distributed configuration of NCSs
Sensor node
Actuator node
Controller node
Communication links
A Communication Network
Controllers' neighboring area

Figure 9 Central, decentralized, and distributed controller arrangement of network control systems [5]

that SDs have to handle monitoring and component management tasks as well.

TD standardization takes the approach of choreography service composition for successfully providing scalability in the WoT world [16]. In contrast to service orchestration where composed services control other services, choreography service compositions let distributed devices describe their capabilities and interactions by themselves. This distributed approach allows single TDs to expose their own or consume external WoT compatible functionalities. A SD has the purpose to specify application properties of a WoT mashup which consists of single Things. To allow adaptation, a SD has to interact with a controlling unit. In the same way as a WoT mashup can keep a SD central, decentralized, or distributed, the adaptation controlling unit faces equal design decisions. Moreover, the best location for the controlling unit which interacts with the SD also depends on the application.

The standardized SD proposal for WoT mashups of this work follows a the centralized approach. The issues of distributed and central system approaches of the previous paragraphs justify this reason in the following way. Figure 9 (c) shows with the distributed arrangement of controllers that there is more communication between controllers compared to the centralized approach. More communication caused by the required consent state of the controllers produces more controlling and computational efforts [48]. WoT mashups depend on IoT devices with constrained computing power. This motivates the decision to keep the SD on a central component together with the controlling unit. A mashup which keeps the SD together with the controlling unit will use more resources and thus require more computing power. At the same time, the approach prevents other Things which manage TDs from additional computational efforts. This approach exploits the centralized benefit of requiring only one computational stronger component. The modular design of WoT Servients take scalability issues into account.

A possible compatibility of a SD with the WoT stack provides the ability of interoperation with other SDs. This capability therefore takes scalability issues already into account. At the same time, the utilization of the distributed approach with its implicit scalability and reliability benefits is not a requirement of this work.

3. Methodology

The methodology chapter explains two heavily used methodology concepts of the WoT field. It moreover introduces data formats and software tools that contribute to the SD proposal and implementation of upcoming chapters of this work.

3.1. REST Based Communication

The representational state transfer (REST) is an architectural style for building Web services. First introduced in the doctor thesis of R.T. Fielding [26], the REST architectural style continues to evolve and hold significance in modern Web architecture designs [50]. Web architectures and services which implement or use REST are called REST-ful. Due to features such as simple distributed applicability and interoperability, REST-ful Web services have substantial share in the success of today's scalable Web. The architectural style enables to access and manipulate textual representations of Web resources through a limited set of actions.

Resources in the Web are documents with Uniform Resource Locators (URLs). A URL serves thereby as an address for each resource. It is possible to request a Web resource from a REST-ful Web service. The REST-ful Web service can handle different types of requests for accessing and modifying a resource. Most REST-ful services are implemented with the HTTP communication protocol and its GET, PUT, POST, and DELETE request types. If a Web service counts as REST-ful, it implements default behaviors of the architectural style. Default functionality specifies the ability to receive, delete, modify, or send data to a resource. Taking the HTTP protocol as an example, the GET request returns a requested resource, the PUT request updates or creates a resource, the POST request allows to send data to the resource, and the DELETE request deletes a resource. Data formatting of the of messages depend on the decisions of the REST-ful web service developer. REST only specifies the abstract architectural style of resource accessibility and modification.

Since REST-ful Web services play an important role of the Web, they also have equal importance in the WoT [27]. REST-ful principles affect the design decisions of TDs. The interaction property type of a WoT Servient implements two types REST-ful behavior if it is writable and observable. It is possible to read a Thing property if it is observable. If the property is writable, the write method allows to modify the property value. Invoking an action besides that calls of a resource and sends input data. Since TDs are designed to comply with REST-ful requests, design decisions for SDs can respect and leverage this underlying concept.

3.2. Semantic Web

The purpose of the semantic Web is the comprehension of semantic documents and data through machines [51]. The word semantic refers to explain the meaning of something. Web semantics do not describe resources and their functionality. They map a meaning for a resource to a context and solve the problem of multiple meaning of a resource. The following example clarifies the problem.

Having a HTML resource with the name tag and a verbatim value such as lion, it is not clear for a machine whether this resource means an animal, the chocolate bar, or a movie called lion. People have the ability to resolve the meaning of a resource through context awareness. It takes people years to correctly learn to differentiate between multiple meanings of some words. Machines do not have the capability to differentiate meaning without additional information. Therefore web semantics specify a well defined meaning for a resource by defining a context.

Semantic information may surround a HTML tag or element. This information links the resource into a context of specified vocabulary. One of the largest structured data repositories of shared vocabulary maintains the website schema.org [52]. The website creates, maintains, and promotes schemata for structured data to facilitate the share of vocabulary. It is moreover possible to use the schema vocabulary in different data formats. Resource Description Framework (RDF) and JSON-LD data formats which are described in section 3.3 help web crawlers and other services to manage related data. With the help of semantics, it is possible to automatically seek related data of a resource and provide improved user experience.

Semantics find application in WoT TDs. Because of heavy usage of REST services and their dependence on URIs, WoT TDs require a way to correctly indicate the meaning of the huge amount of different Servient capabilities. JSON-LD keys in line 2 of Listing 1 show an example of semantic notation with the $@context$ key. The context key provides and at the same time links the TD to the W3C WoT TD semantic context vocabulary. Thing, properties, actions, events, and security schemata for instance belong to this TD context. Another frequently used semantic notation inside WoT TDs is the $@type$ key. This key lets the consumer of the TD know which semantic type the TD of the semantic context vocabulary of TDs implements. Knowing about the Thing schema, a consumer is able to look up the W3C definition of a Thing. As described in the previous paragraph, the methodology of semantics used with TDs clarify the meaning and structuring vocabulary of the TD.

3.3. Data Types

WoT TDs specify their structure by using JSON Schema [53]. JSON Schema is based on the well known JSON data format. The TD Testbench which will be introduced in this chapter uses properties of JSON Schema structure to automatically generate and validate requests and responses to and from TD Servients. The following subsections provide all necessary details about different variations of the JSON data format which represent the underlying data types of TDs.

3.3.1. JSON & JSON-LD 1.1

Java Script Object Notation (JSON) is a human readable, open standard, and language independent data format [54]. It consists of multiple or zero key value pairs within an object structure. The data type of the key is string whereas object, array, number, or string data types of values represent. Furthermore single types such as false, null, and true complete the value data type repertoire. Figure 2 shows a valid JSON data example. The example marks keys blue and shows different value types. The first value type is a string and the second value type shows an object. Values inside the object value show all other possible value types.

Java Script Object Notation-Linked Data (JSON-LD) extends existing JSON in a way that JSON becomes accessible as Linked Data [30]. Linked Data [55] with the purpose to help the semantic web connect data uses links so that persons or machines can explore the web of data automatically. JSON-LD extends original JSON grammar and defines specific key attributes and values. Next to machine accessible data extensions to the JSON format, JSON-LD enables to store Linked Data in object storage engines. Listing 4 represents a JSON-LD sample. Line 2 and 3 of Listing 4 show semantic annotations. The first annotation sets the JSON-LD document in context of a person. The `@id` key assigns the document a unique resource.

Listing 2: JSON data

```
{
    "key": "value",
    "key2": {
      "key3": ["value2", null, true],
        "key4": 10
    }
}
```

Listing 3: JSON Schema

```
{
  "type": "object",
  "properties": {
    "Country": {"type": "string"},
    "City": {"type": "string"},
  },
  "required": ["Country", "City"],
  "additionalProperties": false
}
```

Listing 4: JSON-LD data sample [9]

```
1 {
2   "@context": "https://json-ld.org/contexts/person.jsonld",
3   "@id": "http://dbpedia.org/resource/John_Lennon",
4   "name": "John Lennon",
5   "born": "1940-10-09",
6   "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
7 }
```

WoT TDs use JSON-LD instead of JSON due to semantic extensions and Linked Data features. TDs have higher flexibility and compatibility with standardized JSON-LD utilization. Since JSON-LD sets the underlying data format of TDs, there are verification tools which validate TDs and thereby JSON-LD on correctness. The `thingweb-playground` tool for instance validates TDs [56]. Before final TD validation the tools checks for JSON, JSON Schema, and JSON-LD correctness. The next subsection introduces details of JSON Schema which specifies structure in regular glslinkjsonJSON. Figure **??** provides an picture of the JSON-LD validation tool.

3.3.2. JSON Schema & JSON Schema faker

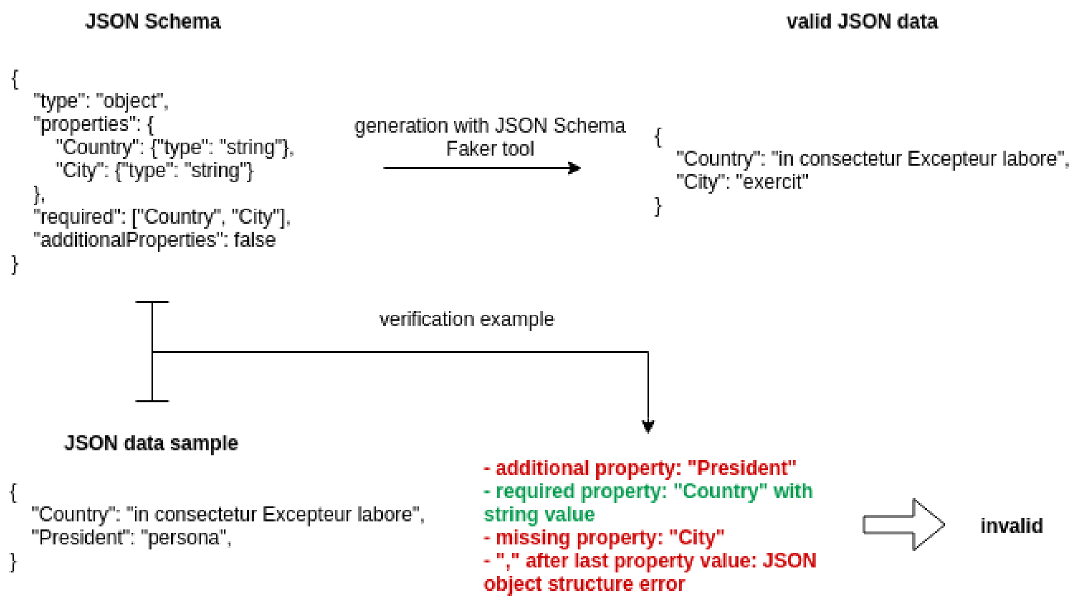

JSON Schema
{
"type": "object",
"properties": {
"Country": {"type": "string"},
"City": {"type": "string"}
},
"required": ["Country", "City"],
"additionalProperties": false
}
generation with JSON Schema Faker tool
valid JSON data
{
"Country": "in consectetur Excepteur labore",
"City": "exercit"
}
verification example
JSON data sample
{
"Country": "in consectetur Excepteur labore",
"President": "persona",
}
- additional property: "President"
- required property: "Country" with string value
- missing property: "City"
- "," after last property value: JSON object structure error
invalid

Figure 10 JSON Schema generation and verification example with JSON Schema faker tool [6]

JSON Schema specifications structure the JSON data format. Predefined key value combinations of JSON Schema impose additional key value rules of the JSON document. JSON Schema core vocabulary describes the structure of JSON objects. JSON Schema enables automated verification and generation of JSON data.

Listing 3 illustrates JSON Schema validation keywords. The `type` keyword thereby restricts an instance to an object with one of the core JSON value types. When dealing with JSON Schema, the term instance refers to a JSON document that is validated. A JSON document which describes structure refers to the term schema [53]. Listing 3 represents a schema. A valid instance for the schema of Listing 3 is a JSON object with a properties field. This properties field must have the required country and city keys. Moreover the schema specifies that it is not possible to name additional property keys inside the properties value field. A valid object may have other key values around the mandatory object structure though.

Next to validation possibilities, JSON Schema provides JSON object generation techniques. These techniques generate a valid JSON object based on the given schema. The upper part of Figure 10 shows how the JSON Schema faker tool generates a JSON instance. Figure 10 uses the JSON Schema of Listing 3 for JSON instance creation. The JSON Schema faker of Figure 10 moreover provides JSON Schema validation methods. The bottom part of Figure 10 shows an verification example of a JSON instance. The instance is invalid due to the red marked JSON Schema disruptions. Validation as well as verification techniques of JSON Schema can be used to automatically test data of REST compatible requests.

3.4. Software Tools

This work relies on the following software tools. It is recommended to read the subsections one after the other due to the fact that they are explained according to their dependencies.

3.4.1. eclipse/thingweb.node-wot

The eclipse thingweb node-wot project [57] is a first WoT scripting API implementation written in Node.js [58]. It is important to note that node-wot serves as a first implementation and does not claim perfect efficiency, security, and coding style. It allows to implement a WoT Servient object. Node-wot supports HTTP, HTTPS, CoAP, CoAPS, MQTT, and websockets [34] transport protocols. Currently there is only JSON and plain text communication media type support. Nevertheless node-wot is under active development. Many additional protocols and other media type support will extend the current version. Since the project is open source, contributors can extend the supported or create new protocols by implementing the `ProtocolClient`, `ProtocolClientFactory`, or `ProtocolServer` interfaces. It is not mandatory to use Web protocols. New media types need additional implementations of the `ContentCodec` interface.

After setting up node-wot, it is possible to import node-wot data structures. Listing 5 provides a Javascript code sample of a node-wot Servient implementation. Line 1 of Listing 5 imports the WoT Servient class. Calling the Servient class returns a Servient object. With the Servient object indicated in line 5 of Listing 5, it is possible to add desired protocols to the Servient object. This can be done by calling object methods such as `addServer` or `addClientFactory`. The `addServer` method with a HTTP server class as input equips the Servient object with the capability to expose interactions through HTTP. Line 6 and 7 of Listing 5 illustrate this protocol addition. Adding the HTTP client factory class allows the Servient itself to act as a Thing client. When starting a Servient object by executing its start method, node-wot provides a WoT object with scripting API functionality. Methods of this WoT object implement concepts such as registering, discovery, exposure, and consumption of Servients with TDs. The produce method of the WoT object of line 11 of Listing 5 for example exposes the Servient with the name `mything`.

Listing 5: Javascript code sample of eclipse thingweb node-wot Servient implementation

```
1  var Servient = require('thingweb.node-wot/packages/core').Servient;
2  var HttpServer = require('thingweb.node-wot/packages/binding-http')
       .HttpServer;
3  var HttpClientFactory = require('thingweb.node-wot/packages/binding
       -http').HttpClientFactory;
4
5  let srv = new Servient();
6  srv.addServer(new HttpServer(8081));
7  srv.addClientFactory(new HttpClientFactory());
8  srv.start().then(WoT=>{
9
10     console.log('* started servient');
11     let thing = WoT.produce({
12         name: "mything",
13     });
14     thing.addProperty("temp", {
15         type : 'number',
16         writable : false,
17         observable : true
18     }, 0);
19     .
20     .
21     .
22 }).catch(err => { throw "Couldnt connect to servient" });
```

The returned object of the produce method is able to add, remove, or handle TD interactions and other JSON keys of the TD of the exposed Servient. The Thing object has methods for adding, removing interactions and respective handlers. The Thing object of line 14 of Listing 5 utilizes the `addProperty` method for creating a new TD property. If a WoT object consumes a TD, the returning consumed Thing object has the capability of calling interactions of the consumed Thing.

3.4.2. Thing Description Testbench

The TD Testbench originates from a master's thesis about semantically-extended Things Descriptions for designing and testing IoT devices [7]. It tests Things by using their TD and relies on node-wot [57] version 0.3.0. The Testbench itself is a WoT Servient implementation that exposes a TD. Since every node-wot WoT Servient has a TD, the Testbench is able to test TDs by interacting with other WoT Servients. After reading, writing or invoking an interaction, the Testbench verifies if received return values match the description found in the testing TD. Figure 11 and 12 illustrate the Testbench and its testing procedure respectively.

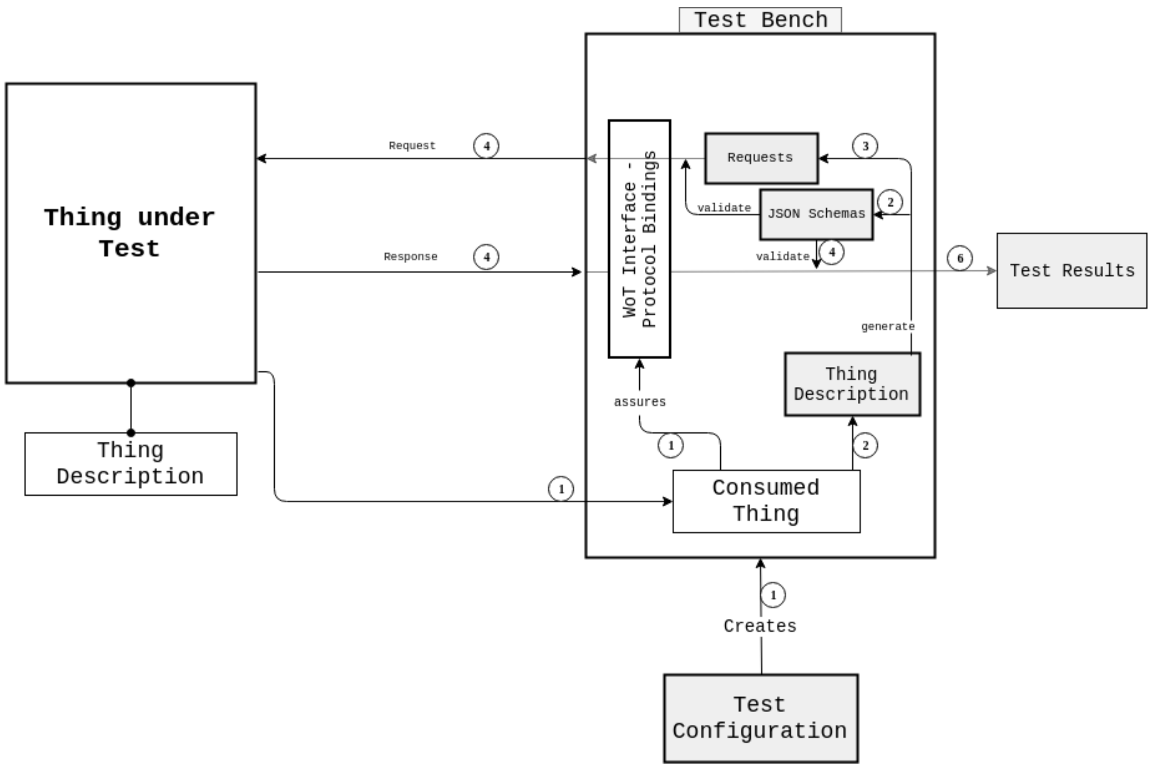


Figure 11 Testbench component overview [7]

The TD Testbench starts by reading in a configuration file. Information of this file configures ports and the name of the Testbench Servient for later communication with other Things. Moreover it is possible to specify the path of requesting data for later interactions. Is is required to specify the path to the TD that is going through the testing procedure. The configuration file moreover allows to set the path of where to store resulting test reports. Lastly, the configuration file reads in testing properties such as the number of repetitions and testing scenarios.

After reading in all configuration values, the Testbench Servient launches and consumes the provided TD. Figure 11 illustrates the tested Thing and the Testbench on the left and right side respectively. Before calling the testing action of the Testbench, the initialization action sets up all mandatory objects for the subsequent testing process. Affected fields of the initialization process inside the Testbench of Figure 11 are requests and JSON schemata fields. Requests formulate input data for all testing scenarios. JSON schemata help to validate exchanged data schemata. The setting up all mandatory prerequisites by calling the initialization, the testing action invokes the testing procedure. Figure 12 provides intermediary steps of the testing procedure. After the testing procedure completes, the Testbench stores test results in the configured location.

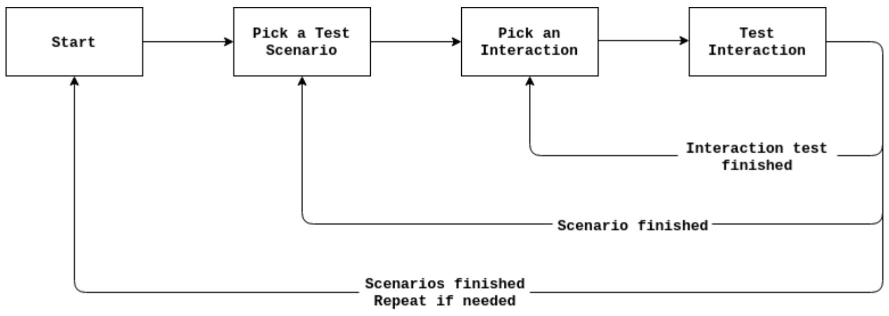


Figure 12 Testbench testing scenarios with repetition [7]

Figure 12 shows possible repetition, scenario and interaction options of the testing procedure. Request input data of the Testbench affects the top right interaction loop of Figure 12. By taking a writable and a non-writable property of a Thing as an example, it is possible to depict differences of the interaction and scenario loop of Figure 12.

The testing procedure of a Thing always starts with the first testing scenario. In the case of testing a non writable property, the testing procedure reads the property and verifies its response. Afterwards the scenario finishes. In the case of testing an writable property, the testing procedure reads, writes, and reads the writable property. All these interactions on the same property belong to a single testing scenario. Testing another property of the same TD would cause the scenario number to increment. The repetition number allows to run equal scenarios again. This feature of the Testbench enables to test TD interactions with regard to temporal behavior.

3.4.3. Postman

Postman is an API Development Environment (ADE) [8]. The Postman ADE provides enhanced API development. Figure 13 shows the main services that the Postman tool provides. First, the Postman application interface allows to enter a requesting URL. The Postman echo API server responds to the request if it hosts the requesting URL address. Afterwards, the Postman application interface receives and displays the returned response. Postman supports HTTP requests and various server endpoint implementations. Next to HTTP methods, the Postman echo server handlers authentication mechanisms, form data vari-

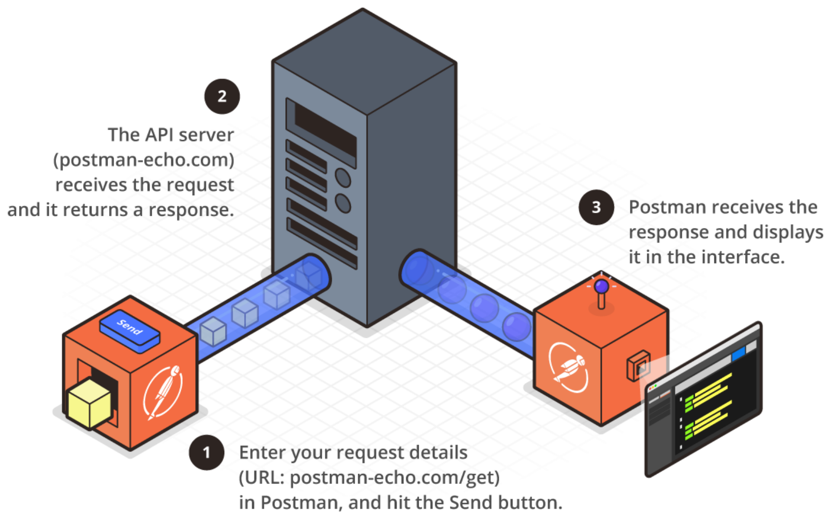


Figure 13 Anatomy of a Postman request [8]

ations, cookie handling, and other utility endpoints. Client applications can therefore use the complete Postman tool set for API protocol and mechanism testing.

Nevertheless, this work mainly uses the Postman application interface for WoT Servient interaction testing. This is due to the fact that requested URLs target self developed Servient implementations. Postman provides the ability to gather and store API requests inside collections. It is moreover possible to execute requests inside a collection sequentially. This capability of the Postman client interface enables automated and sequential configuration and initialization requests for setting up the Testbench as a service with required input. The Testbench documentation specifies example requests for setting up and running the Testbench as a service. Here, Postman provides collection documentation through automatically created websites. Different collections shared in a Postman workspace moreover help to distribute and make Testbench requests accessible.

A typical interaction flow between the Postman client and the Testbench could be implemented in the following way. First the Postman client starts with requesting the configuration data. The write request allows to overwrite default configurations of the Testbench. Afterwards, it is mandatory to set the Thing under test TD with another write method. By invoking the initialization action, the Testbench extracts JSON Schema out of the given TD. At the same time, the Testbench produces valid fake data. It afterwards exposes the generated data. If a user wants to change or use his own testing data, it is possible to do so. When all testing parameters fit the users needs, requesting the test TD action of the Testbench starts the testing procedure. Again, the Testbench exposes the test report in the end. By calling the report with a read request, the Postman client is able to inspect the report.

3.5. Hardware Devices

For running WoT Servients, this work utilizes Raspberry Pi 3 devices [59]. The Raspberry Pi Sense HAT add-on board equips Raspberry Pi devices with temperature, humidity, and other sensors next to a LED matrix. Each WoT Servient implemented on the Raspberry Pi devices exposes temperature properties through TDs with the help of the underlying hardware. A wireless router connects the WoT Servient devices in a local network. A laptop runs the Testbench and interacts with the system mashup of Raspberry Pi devices. The hardware of this work surpasses minimal requirements of running utilized software. This convenience ensured trouble free interplay of the software components. Testing minimal hardware requirements of hardware components of a WoT mashup is beyond the scope of this project.

4. System Description Proposal - (Modeling and Description of System Description Idea)

The two main criteria points for modeling a SD is the WoT stack compatibility of the SD and its scalability. This means that first of all a SD should have the ability to interoperate with WoT TDs. Interoperability of the SD with the WoT TD is the first major requirement because every WoT system mashup component contains a TD of its service. Secondly, utilization and applicability of a SD should be scalable. As WoT principles already target the capability of scalability for TDs, TDs again set the central orientation for a SD.

4.1. Integrating a System Description into Web of Things concepts

The following proposed idea of a SD for WoT stack compatible Things is to integrate SD specifications into existing TDs. Doing so provides the first important benefits of the proposed SD. The integration of SD specifications into TDs allow other WoT Servients to discover, fetch, and consume a SD by performing existing functionalities to the connected TD. All thoughts and functionalities of the WoT scripting API could become applicable if the Servient with SD specifications exposes itself. Apart from that, SD would provide compatibility with existing Thing directories introduced in Figure 7. Discovery processes potentially serve for adaptive capabilities of a system. An example is a component failure. Automated replacement would require automated service discovery. Single Thing compatibility therefore has to impact considerations of a WoT SD with regard to goals of this work.

Figure 14 shows the idea of embedded SD specifications into a TD in the center node. At the same time, Figure 14 indicates an example system mashup for a SD implementation. Inside the Figure, there are other four temperature sensor nodes around the center node which together form a system mashup. Each temperature sensor device node runs a WoT Servient which hosts its TD.

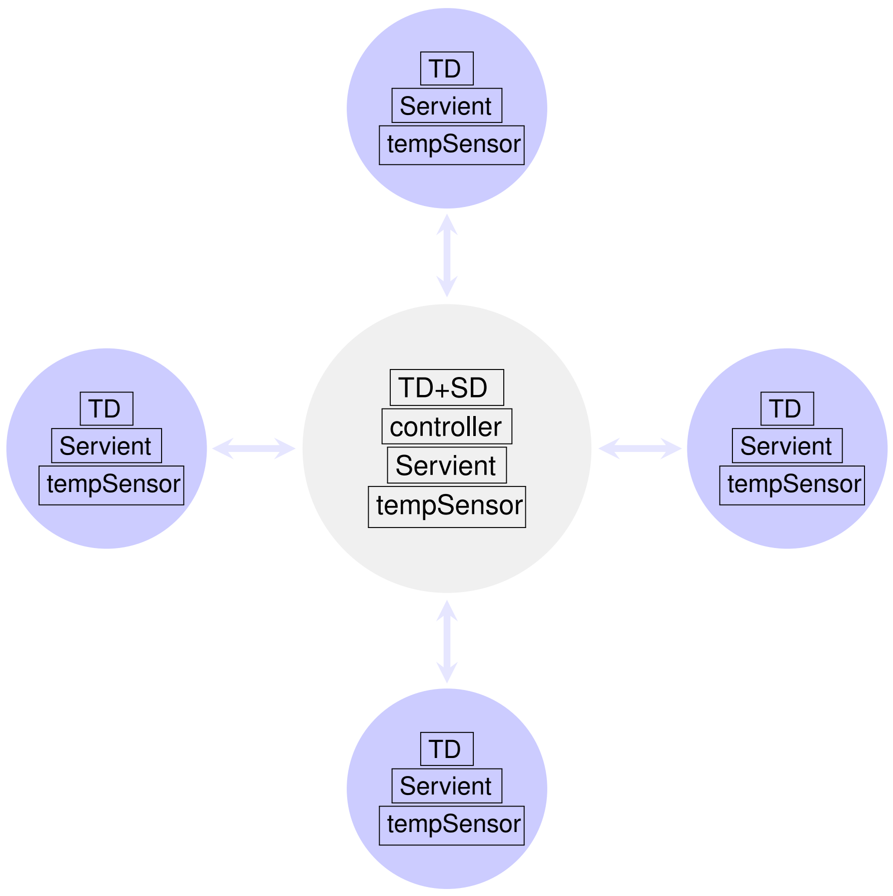


Figure 14 Thing Description mashup example for a System Description

On top of the Servient of the central node, the sensor device uses a controller component. The reason why the central node of Figure 14 contains a controlling element will be introduced in subsection 4.2. When all components of the mashup system provide WoT compatibility through Servient implementations, the central node is able to consume all interactions of system participants. At the same time, the central node with the SD is capable of exposing system functionalities through system dependent interactions to exterior Servients of the WoT. The SD has the purpose of defining all system dependent interactions. In order to provide existing communication solutions between WoT Things, SDs must keep scripting API interfaces and core TD vocabulary.

To meet stated compatibility expectations, SDs may use either JSON-LD or semantic notations to embed additional specifications into existing TDs. This is because of the fact that TDs follow the JSON-LD data schema and semantic methodology. This means TDs are extendible by adding additional JSON keys or semantic annotations. The idea with the proposed SD is to add a new JSON key called `system_name` and describe all other system specific extensions through semantic notations. The reason for this is that the additional JSON key as a TD field name enables easy and early detection of SD specifications. Would SD indicators only depend on semantics, there would not exist an easy way to know whether to parse for semantic system extensions or not.

Apart from the single JSON key which allows to set the SD name, semantic notations mark all other SD components. This is because semantics can leverage predefined interaction types and data structures. The TD manages to abstract communication interactions to types such as properties, actions, events, and links. It makes sense to add semantic notations to interactions to indicate system features. Semantic notations thereby only extend existing interaction types. They do not cause to limit scalability and flexibility advantages of the WoT concept by introducing new interaction types. New interaction types might affect and thereby limit already working standards of single TDs. The semantic notations of the SD of this work leverage existing single TD interactions only. There might be system requirements which demands further new interaction types though. Researching system dependent interaction types goes beyond the scope of this project and represents future work. By adding semantic notations which imply system specifications, it is possible to embed a system property into the interaction properties field name. The same idea gives system events and actions equal possibilities to express themselves.

Another last point about the proposed thoughts on how to model SD interoperability with WoT devices is that it is feasible for a Servient to only host the SD. That is possible if the TD of the system controlling component only contains SD specific information and interactions. When taking the mashup composition of Figure 14 as an example, it is possible to have the TD of the central node only contain system descriptive information instead of additional interaction types about the underlying temperature sensor device. No description about the central temperature sensor device inside its TD means that this device serves as a controlling system element only.

4.2. Central or distributed approach for System Descriptions

This section refers to the management, monitoring, and adaptation of system components with the help of a SD. The question comes with the problem whether the SD should be distributed or kept in a central position when sharing system information. In other words, will there exist one orchestrating, scheduling, and controlling component which manages the system integrity, or will every device maintain knowledge and system awareness. Figure 14 already indicates that the SD distribution stays within only one component. Therefore, the proposed solution of this work to the question whether to apply a SD in a distributed way follows the central approach.

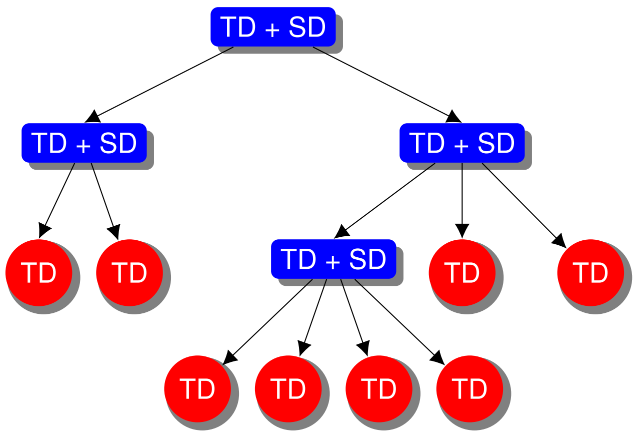


Figure 15 System Description tree abstraction to illustrate scalability

By considering the central approach, there is the advantage to schedule and orchestrate system components within only one component. This non complex approach drastically facilitates the implementation of management and monitoring of the whole system status. In comparison to the distributed approach of SD distribution, the central approach requires one controlling loop which constantly fetches and monitors system information. A central scheduler component of the system can have a modular design and allows simple replacement strategies when system components fail due to connectivity or functionality problems. Modular design of the controller provides a first solution to scalability problems within the central approach of SD custody. Drawbacks about central maintenance of the SD are the single point of failure and scalability if there are to many manageable devices per SD. To many controlled elements might exceed the computing power capabilities of the controlling component. Even though scalability might not be as easy to solve compared to a distributed approach of applying SDs, controlling elements allow to consume and manage subsystems of device networks. If SDs make up components of other SDs the central application approach should be able to solve scalability issues.

Figure 15 illustrates a tree structure. Here, the overall system consists of multiple systems with SDs. Leafs of the tree vary between devices with only TDs and devices with Thing and SDs. The system abstraction from Figure 15 moreover indicates a mixture between the distributed and central SD approaches which is possible through WoT concept compatibility. An important factor of the SD arrangement of Figure 15 is the scalability testing capabilities. With the hierarchical organization of SDs, is is possible to virtually or physically compose SD subsystems before adding another layer with a managing SD on top. This SD design allows to grow mashup sizes by TD, SD, or both components. Therefore, multiple scalability testing cases allow testing of the desired growth criteria.

The distributed approach of distributing SDs to all participating system devices compels every system component to manage the SD. Therefore every participating device requires controller element which handles system integrity tasks. High complexity is the price to pay for reaching consensus in a distributed multi agent system. Consensus protocols which handle termination, integrity, validation, and agreement problems set difficult implementation requirements compared to the central application approach of SD. For achieving the advantage not to have a single point of failure, the distributed approach needs at least three participating devices. The main advantage after handling the complexities of a multi agent distributed system is possible horizontal scalability. This is due to the fact that each device implements an autonomous organism which is able to participate in the big system.

As a result of contrasting pros and cons of both SD system approaches, this work follows the central approach. The reason for this is the implementation simplicity and easy integration into existing WoT standardization concepts. Applying Servients and TDs in distributed systems with consensus protocols would require many more extensions to existing WoT concepts rather than semantic annotations inside already defined TD field names. The compatibility of WoT concepts brings additional scalability perspectives to the constrained central approach. The modular design of the controlling component and Servients in general allow Web related and dependent scalability solutions.

4.3. System Description specifications for internal system control

After providing descriptions about how to integrate SD specifications into TDs and clarifying the SD localization approach, internal system control and communication marks the next step in modeling a SD. The goal for internal system control is that the system should have adaptive capabilities which implies the use of component monitoring.

It is vital for this SD approach that the controlling component interacts with the SD. The reason for this is that the SD contains all necessary system information which the controlling unit requires for monitoring and adaptation of the system. The proposed system mashup of this work from Figure 14 keeps the SD together with the controller module on one device. The simple mashup does not require sophisticated modular implementations of the controlling unit. Therefore and due to the goal of a first working application of a SD, this work refrains from additional ideas of controlling unit implementations. With the help of an interval loop, it is possible to poll other devices with periodic requests. Doing so allows the controller to monitor the system status. Going into further details of how to most efficiently solve the system awareness problem is beyond the scope of this project. Now with a runtime which hosts the SD and a controlling unit, it is possible to implement system runtime adaptation.

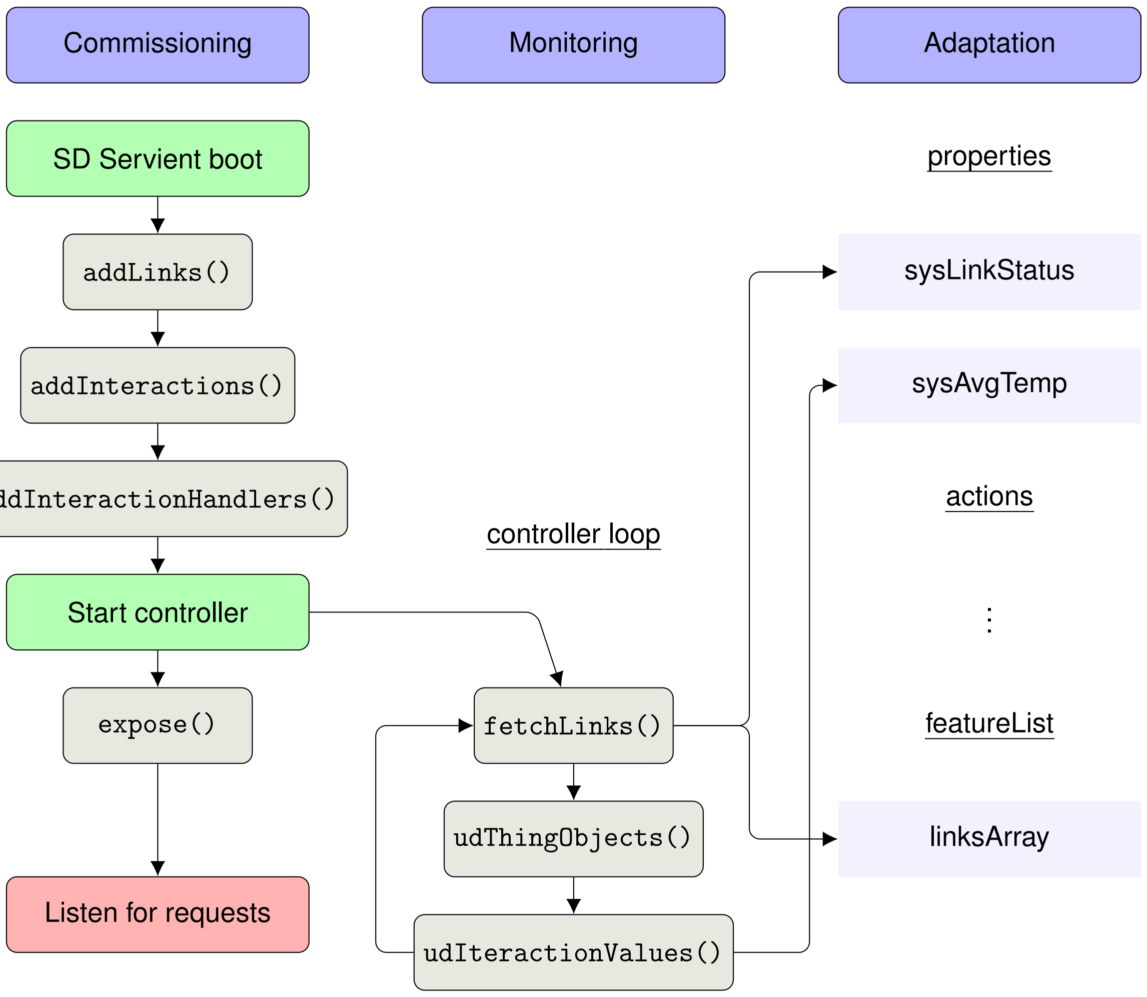


Figure 16 Model of System Description flowchart

Figure 16 allocates mandatory functions and modules of the proposed mashup to three abstraction fields. Commissioning, monitoring and adaptation mark those three partitions. The commissioning column of Figure 16 launches a Servient and the runtime environment of the SD. The monitoring column divides controlling requirements into an execution interval. Lastly the adaptation column illustrates SD interactions for the proposed mashup of this work. With the aid of Figure 16 the next paragraphs outlines the interplay of those components.

To enable communication between the central and all other system components, the SD has to maintain a list of URIs of system devices. An idea is to use another new JSON key next to the `system_name` key. This `featureList` key has the purpose to store URIs to other system component TDs. The reason for not using the already existing `link` field of a TD is that the `link` field can store all sorts of links. It is therefore not clear which links connect system components to the SD. Another possibility could have been to add semantics to links inside the `link` field. This way, the detection of system components would require parsing the whole `link` field. Therefore this work implements the approach with a separated new `featureList` JSON key for linking system components.

The commissioning column of Figure 16 has the `addLinks` function for the purpose of specifying system components by adding URIs. The links are a prerequisite for the controller loop of the monitoring column. The `addInteractions` and `addInteractionHandlers` functions allow to define system interactions and their handler functions. While system interactions heavily depend on particular demands of the system, there is no profound definition of what every system requires. To provide an example though, the array of TD URIs in the SD `featureList` field derives the opportunity to specify system properties which show the link activity status. Additional system actions and events give other interaction examples and options for system management through the SD. Another concrete example interaction could be the action interaction type for starting or stopping the controller loop of the system. Remember that SDs equip system interactions with additional semantics for the controller. This way, the controller knows which system interactions to update and adapt. After adding interaction information, the Servient starts the controller, exposes itself to WoT devices, and listens for incoming service requests.

When looking of the right two columns of Figure 16, the controller loop of the monitoring column starts up when the Servient executes the controller loop. The controlling unit of the SD of this work constantly loops through three functions periodically. This interval functionality manages the adaptive behavior of the system. To showcase adaptive behavior, this work specifies system features derived from the mashup system example of Figure 14. The SD of this application example uses the average temperature property for adaptive behavior. Next to the main average temperature property of the application example, the adaptation column of Figure 16 lists additional properties such as the `sysLinkStatus` and the `linksArray`. The explanation for the existence of these properties is due to differentiation purposes between system control and system functionality. Semantic notations will divide controlling and functionality aspects of the system. Subsection 4.4 provides detailed explanations of how this work implements the distinction in detail.

After outlining important design distinctions and adaptation contingent requirements of the system, it is possible to explain the controller loop functions with the effect on adaptive behavior. Back at the controlling unit, the controller loop starts with the `fetchLinks` function. This function call takes the WoT scripting API fetch function to request TDs. Since the input to the fetch function is an URI, the controller has to accesses the `featureList` field name of the SD. For explanation purposes, it is helpful to assume four URIs in the `linksArray` and an interval function which executes every ten seconds. If some of the fetched URIs does not return a TD, this work uses the system property `sysLinkStatus` for displaying active connections to system components. As this property describes system control, it counts as a system control interaction. The system property `sysAvgTemp` on the other hand counts as a system functionality interaction. The second job of the `fetchLinks` function is to update the `systLinkStatus` property. This update is the first adaptive measure of the controlling unit. Now with the knowledge of active system components, it is possible to consume all TDs of the system components. This part is done by the update function called `upThingObjects`. The name of the function comes from its second task to update single Thing objects after the consumption. Thing objects allow to request single properties of each Thing. This work uses temperature sensor devices to run Servients. All TDs of the devices provide a temperature value as a property. The controller loop `upInteracitonValues` function requests this property of every active Thing object. After accumulation of all property values, the `upInteracitonValues` function calculates the average temperature and updates the system property `sysAvgTemp`. The last function of the controlling loop affects adaptation of a system functionality interaction value. Afterwards the controller loop repeats itself on the next interval call and maintains periodic system adaptation.

The reason why the controller loop calculates the system property of the average temperature value is self-explanatory when looking at computing expenses of the system. There are two ways to provide the average temperature value in the proposed system application of Figure 14. First, there could be a function that requests all temperature values of all system components, calculates the average temperature, updates the system property value, and returns the calculated average temperature. These sequential or concurrent steps would become executed on a property read request. Secondly, it is possible to perform the calculation and update steps inside the controller as proposed in Figure 16. Assume one hundred property read requests within ten seconds. This case would mean one hundred times more computing effort to satisfy client needs compared to the interval property update approach. The interval approach calculates the average temperature value once in an interval of ten seconds. Afterwards, the system responds with the same value for the duration of the interval. The trade off here affects real time deployment of the average temperature property. Within industrial high performance systems, real time deployment of service data is mandatory and requires highly optimized and fast asynchronous implementation solutions of property deployments. Therefore, an interval updating system design might not be sufficient to satisfy system requirements of performance critical systems. In general though, use cases such as smart home environments have no drastically varying temperature values within seconds of time. For this general case, the proposed system application of this work provides and example application.

As one last point regarding SD application design, developers have to think whether interactions from the descriptions depend on each other. Dependencies of single Thing properties could affect the performance by limiting asynchronous data accumulation inside the system. Sequential implementations must be used to implement dependent interaction deployment. Inside the simple system design proposed in Figure 14, sensor properties do not depend on each other. That is the reason why this work follows an asynchronous data accumulation approach for offering the average temperature as a property.

4.4. System Description modeling for WoT compatibility

As a next part, there is the explanation of how to extend existing WoT stack implementations. Next there is the question about what specifically will be used to extend existing WoT stack implementations and achieve the desired functionality of handling SDs. This last section aims to explain those two last missing considerations about a SD.

As already mentioned, semantic annotations will be the way to integrate system related information into existing WoT stack concepts. An already existing repository which gathers concepts of semantic context is the schema.org repository [52]. Schema.org provides a `SoftwareApplication` schema which defines schematic system vocabulary. As a first indicator inside the JSON-LD form, the `@type` key embeds the software application schema context into the description. Listing 6 illustrates a TD with SD semantics. The software application schema context provides the application category JSON key which describes the application type. A consuming client of the SD is able to parse this keyword. As a result the client decides whether to parse for further semantic system information or not.

The first mandatory information inside the SD sets the `featureList` JSON key of the description. The key derives from the software application context and allows a value type of text or URL. The proposed SD makes use of the text value and uses a comma separated list or URLs to other TDs. Only the TDs URLs inside the `featureList` key reference Things that belong the the system composition. The controller is thereby able to parse the `featureList` URLs. The example description of listing 6 provides additional "mediaType" and "rel" attribute information for the first system URL. Reference URLs of component TDs let the controller fetch and process system management tasks. As a last SD specification, there is the `appcliationSubCategory` key of the application software context.

Listing 6: JSON-LD 1.1 Thing Description sample with System Description semantic annotations

```
{
    "@context": ["https://w3c.github.io/wot/w3c-wot-td-context.jsonld",{
      "SoftwareApplication": "http://schema.org/SoftwareApplication"}],
    "@type": "SoftwareApplication",
    "name": "tempSensor",
    "applicationCategory": "SystemDescription",
    "properties": {
        "sysLinkStatus": {
            "applicationSubCategory": "systemControl",
            "writable": false,
            "observable": true,
            "type": "number",
            "forms": [{
                "href": "coaps://mylamp.example.com:5683/status",
                "mediaType": "application/json"
            }]
        },
        "sysAvgTemp": {
            "applicationSubCategory": "systemFunctionality",
            "writable": false,
            "observable": true,
            "type": "number",
            "forms": [{
                "href": "coaps://mylamp.example.com:5683/status",
                "mediaType": "application/json"
            }]
        },
    },
    "featureList": ["https://servient.example.com/things/tempsensor1",
        "https://servient.example.com/things/tempsensor2"],
    "links": [{
      "href": "https://servient.example.com/things/tempsensor1",
      "rel": "controlledBy",
      "mediaType": "application/td"},{
        "href": "https://servient.example.com/things/tempsensor2"},{
        "href": "https://servient.example.com/things/tempsensor"
    }]
}
```

This field is used to differentiate between interactions which either serve system control or system functionality. Listing 6 applies system interaction semantics only to the properties interaction type. These differentiation semantics for interactions can be used in the exact same way for system actions and events interaction types. The reason for dividing system control and functionality interactions allows the controller and client to distinguish and derive new information about the system. A client that consumes a SD might not be interested in system control but functionality whereas the controller requires system control information in the first place.

One major benefit of the proposed SD is that regular WoT Servients are capable of consuming and interact with the specified SD. This is due to the fact that the SD is based on regular TD concepts. Nevertheless the new additional information permits new ways of interacting with SDs. Scripting API functions such as expose and consume need to understand and process the system information. This work suggests the implementation of extended scripting API methods which improve dealings with SDs. It is important to say that the proposed additional extensions are not vital for the SD to work but optimize implementation challenges and system performance.

Figure 17 shows system aware and TD extending scripting API functions. This first type of functions affect the discovery and interaction concepts of WoT devices. The other upcoming function proposals for the

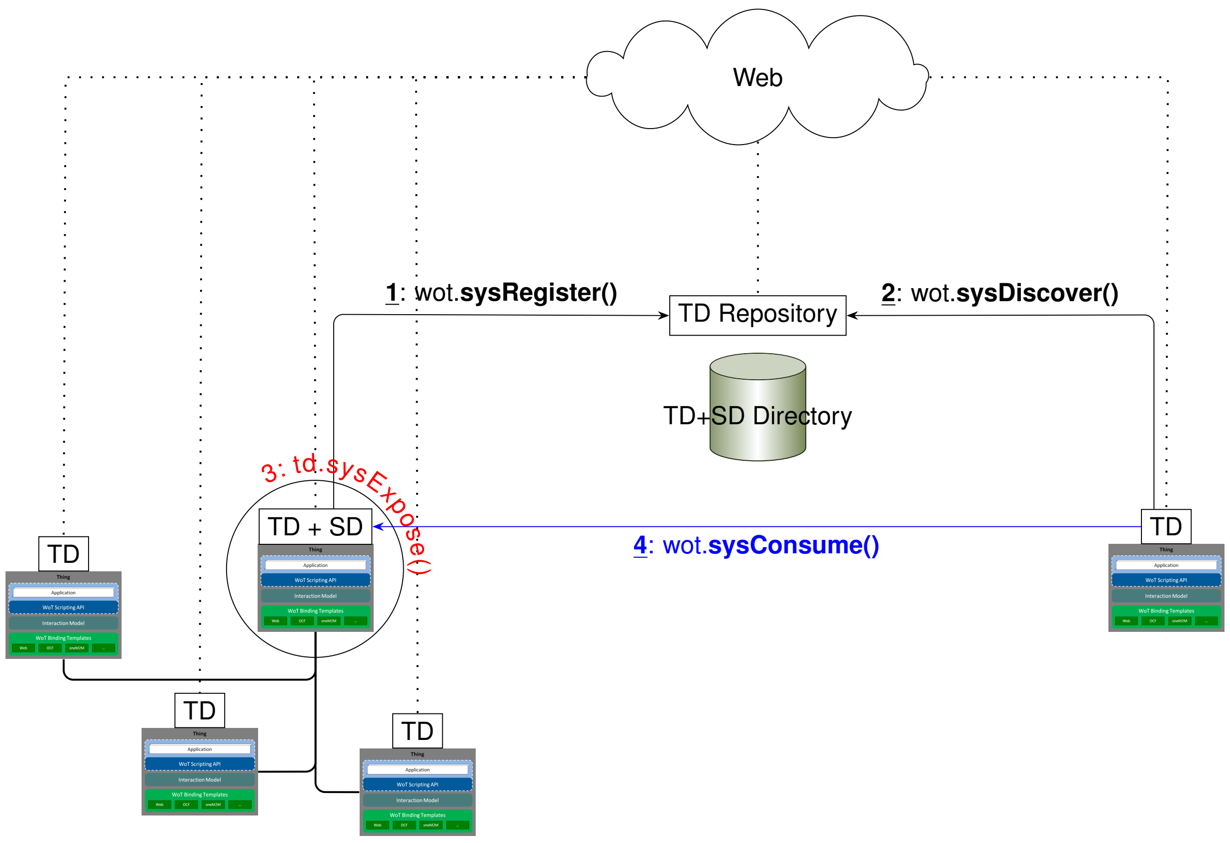


Figure 17 System Description architecture example with adapted scripting API methods

WoT stack aim to facilitate and optimize SD implementations. The first proposal is th extension of functions from Figure 17. The Figure shows the classical process of TD registration, discovery, and consumption. The only difference is that the Figure mentions system related function names. An additional `sysRegister` function for the WoT stack could deliver semantic system add-ons of a TD to a TD repository. This work does not propose an optimized implementation approach of how to store SDs in a directory. The reason for this is the scope of this work and the capability that a registry solution for TDs exist. Therefore it is possible to store SDs as well. Next, it is possible to discover a SD through the `sysDiscover` method which potentially parses and processes system semantics into optimized structures for later consumption. After the exposure of a SD through an additional system exposure function, the last WoT stack extension proposal sets the `sysConsume` method. These suggested method extension could resolve proposed SD information. No other methods and concepts of the WoT stack face changes with regard to system development with basic WoT concepts.

Regarding the facilitation and optimization of SD implementation, it is helpful to have functional capabilities which allow to fetch and consume multiple TDs by one function call. The suggestion here is to extend the WoT stack with `multiFetch` and `multiConsume` methods which take lists as inputs. For this proposal respectively, `multiFetch` requires a list of TD URIs and `multiConsume` takes a list of TDs. These methods help the controlling unit of the system to fetch multiple links and consume many component TDs.

4.5. System verification through System Description testing

Due to the fact that this work supports the development of a TD testing tool, it has the possibility of including existing TD verification concepts to the proposed SD. The SD adaptation capabilities could profit of system component verification. The reason for this is that verification ensures detection of unreliable component features. In the end, the whole adaptation of the SD could becomes more reliable due system internal verification processes.

Again as with the suggestions for extending scripting API methods with regard to SD specific characteristics, the application of SD verification is not mandatory for the SD to work. As the existing `testThing` method of the Testbench should theoretically be applicable to the SD, it is also possible to think about a SD testing method. A SD testing and verification method could include single TD testing methods on system components with TDs. Since the SD proposal addresses scalability and adaptivity requirements, SD verification methods should not interfere with design principles that affect scalability and adaptivity characteristics. An internal SD verification method could ensure safer system commissioning as well as runtime.

5. Implementation

As already indicated in previous chapters, the implementation part of the work divides itself. The first section refers to Testbench development. Updates, extensions and improvements of Testbench features tackle problems of current WoT mashup developers which contribute to TD standardization development. Secondly, this chapter focuses on the implementation of the proposed SD standard and system runtime adaptation of WoT mashups through a SD.

5.1. Thing Description Testbench

This section describes implementation measures which keep the TD Testbench [7] up to date on latest TD developments and specifications.

Since TDs are under development and standards change within frequent updates, TD testing frameworks have to adapt to updates as well. The first implementation of the TD testing framework Testbench is compatible with the 2017 TD working draft specification [60]. The updated Testbench provides TD compatibility with the 2018 TD working draft specification [2]. Extensions and improvements on usability of the Testbench implement a testing framework as a service. REST compatible requests allow to configure and set up testing procedures of the updated Testbench. REST interactions help to outline the complete Testbench workflow.

The first implementation part of this work served for familiarization, implementations, and understanding of WoT concepts. Moreover the updated Testbench contributed to SD commissioning verification with its testing procedure and `testSystem` method.

5.1.1. Testbench update

Before explaining implementation details of all updates of Testbench features, the following list summarizes all affected Testbench characteristics. At the same time, the list serves as an overview of consecutive development steps:

- Global dependency imports of Testbench files
- Latest node-wot version integration
- Log message alignment with node-wot logs
- Schema extractor for new Thing Description standard
- JSON Schema faker tool integration
- Updated testing reports generation and assembly
- New configuration options
- Testable Thing under test Servient implementation
- Documentation on Testbench system requirements, installation, and usage
- New testSystem method for System Description verification

The first and general issue was to create compatibility between different versions of the Testbench and TD specifications. It was an ongoing issue to keep versions of the scripting API, TD specifications, and node-wot synchronized. This factor was mandatory for running the Testbench smoothly. The reason why version irregularities affected the development procedure was that groups in the agile standardization group lacked behind. To provide quick solutions, it was necessary to implement transformation functions for TD version conversions. These translator functions could be used by other developers with similar conflicts.

The implementation of the updating process started with setting up the environment to run the old version of the Testbench. Since the first node-wot update did not work with latest TD specifications, a first translation function translated new JSON interaction forms back to old forms. Hence, the underlying node-wot binding templates provided compatible functionality to other node-wot Thing implementations. The translation functions ensured temporary compatibility with new TDs. In the end and before the Bundang plugfest 2018, a new update of node-wot provided TD compatibility with the latest standard published in April 2018 [2].

Listing 7: Old Testbench configuration file

```
{
    "TBname": "test-bench",
    "ThingTdLocation" : "~/Desktop/thesis/tb/",
    "ThingTdName":"MyStringThing",
    "SchemaLocation" : "Resources/SchemasSimpleString/",
    "TestReportsLocation" : "ReportsSimpleString/",
    "RequestsLocation" : "Resources/req-simpleString.json",
    "Repetitions" : 10
}
```

Listing 8: Updated Testbench configuration file

```
{
    "TBname": "test-bench-V1.1",
    "HttpPort": 8980,
    "SchemaLocation": "Resources/InteractionSchemas/",
    "TestReportsLocation": "Reports/",
    "TestDataLocation": "Resources/fake-data.json",
    "ActionTimeout": 4000,
    "Scenarios": 2,
    "Repetitions": 1
}
```

With an updated node-wot version, it was possible to restructure, adapt, and update the Testbench. Only a few points of the itemization list of this subsection require further explanations. Other points of this list are self explanatory and do not need additional attention.

A smaller update was the JSON Schema faker tool integration which gave the Testbench the ability to automatically verify and generate JSON data. The generated JSON data served a input for invoke and write methods. For data generation, the JSON Schema faker tool used schemata of the updated schema extractor function which extracted JSON Schema out of tested TDs. Another smaller update affected the creation of testing report files. For preventing replacement of files with identical naming, the Testbench incremented and appended a counter within the file name of the test report. This way, the Testbench kept all generated testing reports.

A larger update affected the configuration options of the Testbench. Listing 7 and 8 illustrate the old and updated configuration files respectively. In comparison to the old configuration file, the updated configuration file of Listing 8 had a configurable Testbench Servient name and port. File locations of generated interaction input data, Thing under test JSON Schema, and test reports represented additional configuration options. Next to the repetition rate of scenarios, the new configuration file allowed to set the scenario number and an action timeout. The timeout value which had a unit of milliseconds specified how long action interaction test waited for a response. In the case of longer taking Thing actions, the Testbench continued running next interaction tests or scenarios.

Before coming to last extensions of the Testbench upgrade, this work implemented a sample Servient with the latest node-wot functionality. This sample Servient served as the testing Thing under test. All possible interaction type implementations on the testing Servient helped to ensure testing of all interaction types. The testing Servient implementation was added to the final Github repository of the Testbench for providing an example Thing to test [61]. The Testbench with synchronized TD specifications, node-wot, and scripting API versions for the Bundang plugfest was uploaded to the `bundang-v1.1` branch. The master branch of the Testbench represented the Testbench version which was compatible with latest state of the art TD specifications.

The documentation as another extension of the Testbench compromised system requirements, installation, and how-to-use explanations. Additionally, a screen recording of how to use Postman in combination with the Testbench was published on Youtube [62]. The recording walked through an example testing procedure of the example test Servient as a the Thing under test. The link to the video as well as a link to extended documentation through Postman workspaces can be found in the documentation of the Github repository of the Testbench [61]. The Postman workspace of the Testbench publicly shared the description and collection of interactions made inside the video. Here, command line requesting commands as well as Postman requests specified how to interact with the Testbench as a service. Figure 27 of the Appendix A of this work provides the documentation with an example interaction description from the Postman workspace.

The last new feature of the Testbench was a `testSystem` method which was able to test the SD of this work. Even though subsection 5.2.1 explains the implementation of the SD of this work, it is possible to explain the `testSystem` algorithm of the Testbench. The only necessary information is that the SD of this work contains the `featureList` JSON key which maintains URLs to TDs of system components.

The `testSystem` algorithm of the Testbench starts with the `gatherFeatureListLinks` function of Listing 9. Mandatory input for this function is a SD and node-wot WoT Servient object. The function first extracts and adds URLs of the SD to an array. Line 7 and 9 of Listing 9 show this preprocessing step.

Listing 9: Javascript code of recursive gatherFeatureListLinks function of Testbench testSystem method

```
 1 private gatherFeatureListLinks(systemTD:string, WoT) {
 2     var tdData = {}
 3     tdData["systemDescription"] = systemTD
 4     var links = []
 5     let systemTDjson = JSON.parse(systemTD)
 6     if (systemTDjson.hasOwnProperty("featureList")) {
 7         var featList1 = systemTDjson["featureList"]
 8         for (var jj=0; jj < featList1.length;jj++) {
 9             links.push(featList1[jj])
10         }
11     }
12     return this.recurse(links, WoT, tdData);
13 }
14 private recurse(links, WoT, tdData) {
15     if (links.length < 1) {
16         return tdData
17     } else {
18         return Promise.all(links.map((url) => {
19             return WoT.fetch(url)
20         })).then((results) => {
21             for (var j=0; j < results.length; j++) {
22                 tdData[links[j]] = results[j]
23                 let tdjson = JSON.parse(String(results[j]))
24                 if (tdjson.hasOwnProperty("featureList")) {
25                     var featList2 = tdjson["featureList"]
26                     for (var jj=0; jj< featList2.length;jj++) {
27                        links.push(featList2[jj])
28                     }
29                 }
30             }
31             links = links.slice(results.length)
32             return this.recurse(links, WoT, tdData)
33         })
34     }
35 }
```

By passing the array with URLs to TDs of system components, the WoT Servient object, and a dictionary into the `recurse` function of Listing 9, the TD collection process starts. The `recurse` function fetches all URLs of the array for TDs and checks these TDs for `featureList` JSON key existence. Line 19 and 24 of Listing 9 represent this behavior. If the `recurse` function detects another SD, it adds their component URLs to the array. As the function recursively calls itself again, the node-wot Servient object fetches newly added URLs again. In the end, line 16 of Listing 9 returns a dictionary data structure which contains all TDs and SDs of the system. After the `gatherFeatureListLinks` function, the `testSystem` algorithm consumes every TD and SD of the returned data structure consecutively. Lastly the algorithm executes the Testbench `testThing` method on every TD and SD of the system. It thereby passed the same configuration of repetitions and scenarios to all `testThing` executions.

5.1.2. TestBench as a REST service

Similar to subsection 5.1.1, this subsection structures itself with an overview and subsequent explanatory paragraphs. The paragraphs explain the interaction workflow of the Testbench as a service with the help of Figure 18. It is important to mention that calling REST compatible requests in arbitrarily succession is possible. Even with this possibility, requesting a testing reports before running the testing procedure on a TD does not make sense. The following Figure 18 lists interaction possibilities consecutively to clarify usability of the Testbench:

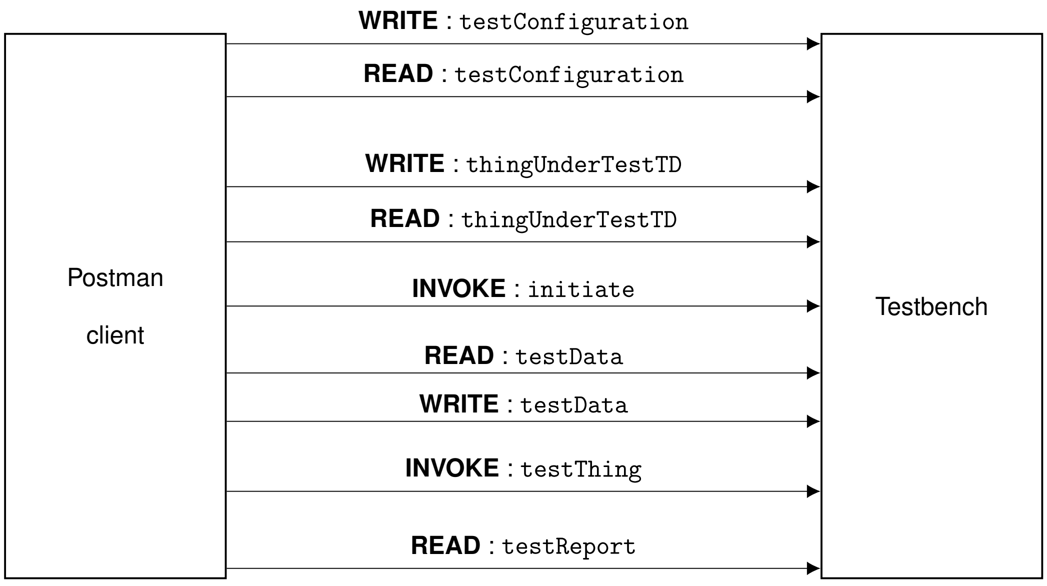


Figure 18 Postman interaction workflow with Testbench as a service

Figure 18 shows the Postman client on the left side and the Testbench on the right side. In between both services, the Figure illustrates scripting API requests in consecutive order. Since the Testbench implements a WoT Servient, it is possible to interact with exposed properties. If for instance a property is writable, a write request allows to modify exposed resources. Figure 18 adds interaction names of the exposed Testbench Servient for showcasing all interaction possibilities between the Testbench and the Postman client. Data of the Testbench configuration, request data, and report files implement writable properties. Generated test data invokes exposed actions of WoT Servients.

5.2. System Runtime Adaptation

The second part of the implementation build upon new definitions and proposals of Chapter 4. Proposals about a SD came chronologically after the Testbench update implementation. Thereafter happened the implementation of the proposed SD. Subsection 5.2.1 describes the SD implementation with a WoT mashup. Afterwards, another subsection explains behavior of the SD mashup with regard to adaptive measures against component failures.

5.2.1. System Description for Web of Things mashup

As suggested in section 4, this work implemented a WoT mashup with a hardware setup of three Raspberry Pi devices. Every device was computationally capable of running more than one node-wot Servient. The implementation followed the mashup proposal of Figure 14 with small deviations. The final WoT mashup composition used three devices with temperature sensors. A Sense HAT sensor add-on board extended raspberry functionality with temperature sensing capability. Every device except one ran a temperature sensor Servient. One device moreover ran a Servient which exposed the SD for the system of three devices. Next to SD exposure, this Servient ran the controlling unit in form of an interval function. This interval function constantly read temperature values of the other devices. This allowed the SD to expose an system average temperature property. Besides temperature value management, the controller inside the SD Servient fetched TDs of system components. This way the SD Servient could expose the system status as a property.

Figure 19 shows the arrangement of devices for the SD mashup on the left side. Regular thick lines connect the SD Servient with other TDs of the other devices. The Testbench on the right side of Figure 19 had to fetch and consume the SD first. Afterwards it could test the activity of other system devices through reading the SD `featureList` field.

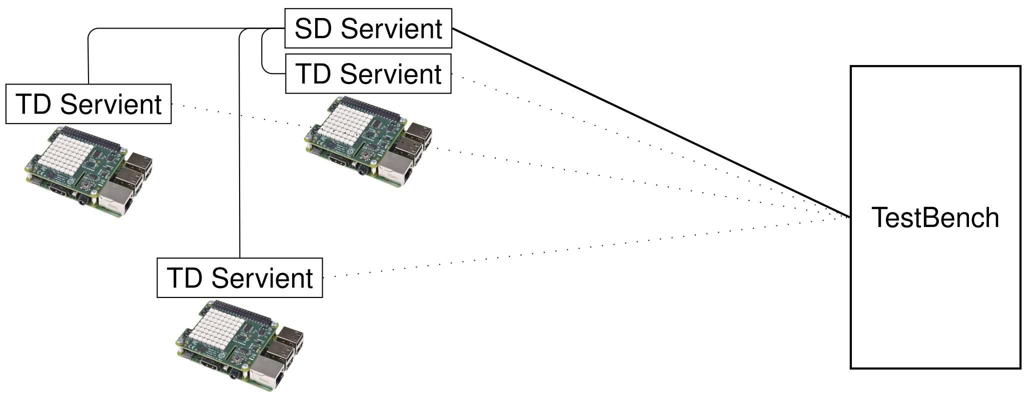


Figure 19 System Description mashup implementation with external Testbench connection

The mashup started with a single device running one temperature Servient. The temperature Servient exposed the temperature sensor value. It requested and updated the temperature sensor value with the help of an interval function. After the implementation of the temperature Servient, the file and environment settings were copied to other system devices. Eventually it was possible to implement and integrate the SD Servient.

The SD Servient exposed the system component status property and the average temperature value of the system. This implementation of the SD was in line with the SD example of Listing 6. Every semantic add-on in the SD of listing 6 was inside the exposed SD. Semantics differentiated between system functionality and system control properties. The SD implementation allowed to test interactions and single components with the Testbench. Interaction testing tested properties of the SD only due to missing other interaction implementations of the SD The Testbench could fetch and consume the SD. Running the overall testing procedure of the Testbench on the SD was possible.

5.2.2. Failure simulation in multiple Things and Runtime Adaptation

After the implementation of the SD mashup, it was possible to provoke measures to showcase runtime system adaptation. The first operation was to disconnect one system component. Disconnection meant stopping the temperature Servient runtime. The design of the controlling unit automatically detected missing system components through the controlling unit. The SD Servient kept an URL of each system component in its `featureList` field. The controlling unit constantly monitored the activity by fetching the URL of TDs of the other devices. This measure allowed to detect not only one but more missing system components. Missing system components affected the exposed SD. The system control property `sysLinkStatus`

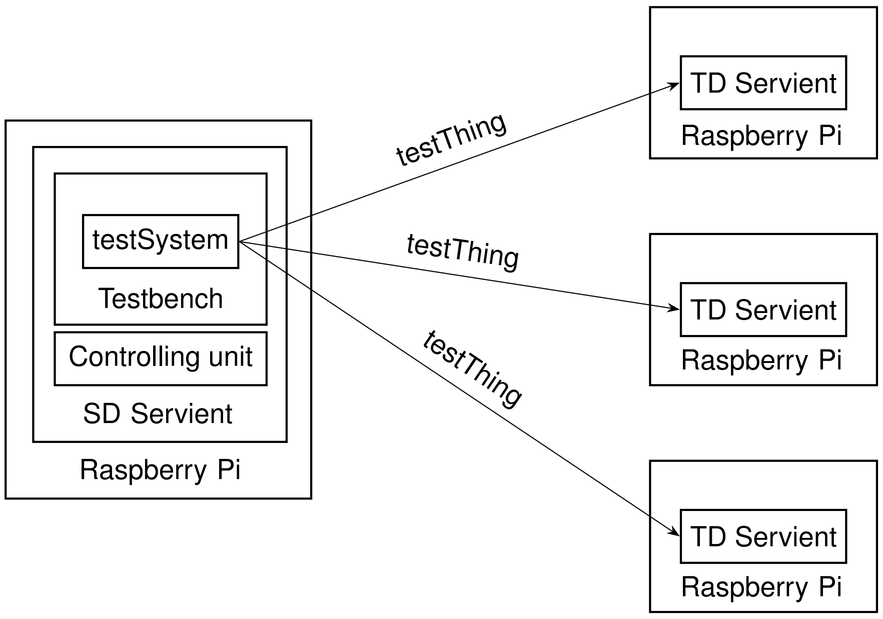


Figure 20 System Description Servient with integrated Testbench for adaptation during commissioning

changed the value depending on the state of system components. This effect ensured runtime adaptation. The missing components also affected the `sysAvgTemp` property. This property of the SD adapted in the way that it calculated its value through temperature values of the remaining components. It is helpful to see the cause of a system component failure in the visualization of application specific properties of Figure 16. A component failure would affect and change the application properties in the adaptation column. The `featureList` array of Figure 16 would stay the same whereas SD properties adapt.

After ensuring adaptation of the SD, the next step to implement was the integration of Testbench methods in the SD Servient. This integration enabled system internal failure checking during commissioning and runtime of the SD Servient. For checking mandatory functionalities of system interactions, this work used the `testSystem` procedure of the Testbench inside the SD Servient. As Figure 19 shows testing possibilities from an external Servient on the SD and its components, Figure 20 shows the SD Servient internal implementation of Testbench methods. Both Figures showcase an application scenario for the `testSystem` method. The difference of Figure 20 compared to Figure 19 is that it displays internal SD testing. Figure 20 moreover indicates the application of the Testbench `testThing` method of the `testSystem` algorithm. As explained in subsection 5.1.1, the `testSystem` executes single TD tests after collecting all system component TDs and SDs. In the end it was possible to call internal SD verification during the commissioning or runtime phase of the SD Servient. When looking at the commissioning phase of Figure 16, the verification happened before the green start controller field.

This new `testSystem` verification procedure ensured runtime adaptation during the initialization phase of the SD Servient. If the testing procedure detected a failure, it would instantly mark the failing state inside the SD. This measure helped and implemented SD adaptation from the beginning of the runtime. It was now possible to expose an already adapted SD. The developed `testSystem` action of the Testbench further equipped the SD Servient with additional reliability. The action which ensured system uptime additionally validated the commissioning phase of the SD Servient.

6. Evaluation

The evaluation chapter of this work focuses on the SD evaluation. It evaluates the SD implementation with regard to scalability. The next sections of this chapter list and report findings and outcomes of the SD evaluation. Afterwards, the discussion section connects outcomes of the evaluation with results of other research.

The implementation chapter clarified that there were no measures implemented for evaluating new add-ons of the Testbench. Improvements of the Testbench relied on advice and well known problems of mashup developers. This work managed to update the Testbench before the 2018 Bundang plugfest. Moreover the updated Testbench will be used by the 2018 Lyon plugfest. Feedback of developers of the Bundang plugfest reached back and can be found in the limitations chapter 7 of this work.

The hardware used for executing measurements for this evaluation consisted of a 2012 Macbook Pro with an Intel Core i7-3520M CPU with 4 processing cores. This computer used 16 GB of RAM and the 64-bit OS version of Ubuntu 18.04.1 LTS. This machine executed all virtually tested commissioning times. The Raspberry Pi devices of the implemented SD mashup ran the Linux raspberrypi 4.6.59-v7 image on a ARM Cortex-A53 CPU with 4 processing cores and 1 GB of RAM. All devices ran the node-wot version 0.4.0.

6.1. System Description scalability evaluation setup

For answering the research question if a SD contributes to scale WoT mashups, this work implemented a measurement script. The measurement script was written in Python. Its purpose was to spawn temperature and SD Servients and measure their initialization times. This section about the scalability evaluation setup explains the structure of the measurement script with simplified pseudocode. Listings 10 and 11 show the pseudocode of the script which allowed to implement the Servient arrangements of Figure 21.

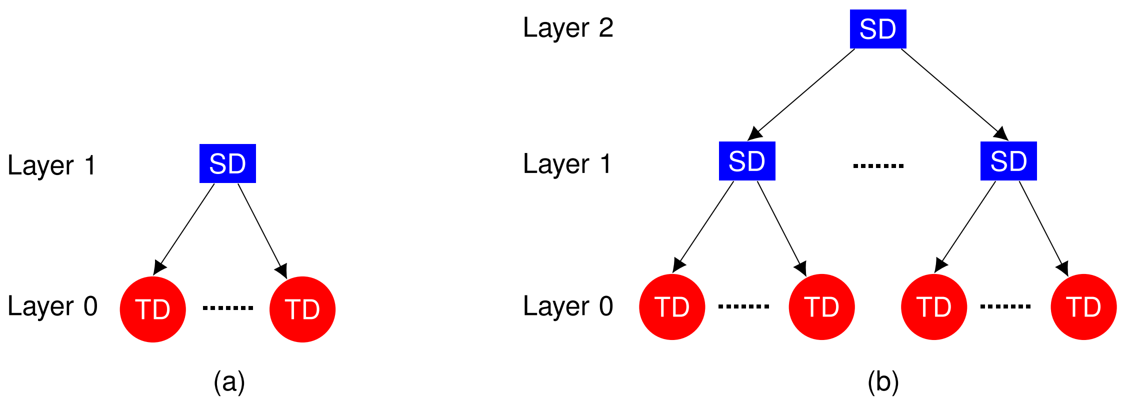


Figure 21 Possible System Description & Thing Description Servient arrangements for scalability testing

The possible arrangements of Servients from Figure 21 allowed to collect data of single as well as multiple SD Servient compositions. The arrangement (a) of Servients of Figure 21 shows the first possible virtual composition option of the measurement script. This arrangement had one SD and a configurable number temperature sensor Servients with TDs. The first input argument of the measurement script indicated the number of TD Servients. Line 5 of Listing 10 shows this setting. By not providing a second argument, the measurement script created a single SD Servient which depended on the specified quantity of TD Servients.

The provided pseudocode of the Listings 10 and 11 as well as the thread model of Figure 22 fit to the Servient arrangement (a) of Figure 21. The pseudocode and the thread model outline required data structures and functions for the implementation of the measurement script. An extension of this model and basic code structure allows to implement the Servient arrangement (b) of Figure 21. Here, it was possible to specify the number of TD as well as SD Servients. When using this option of the measurement script, the script divided the number of TDs by the number of SDs for allocating an equal number of TDs per SD. With a dynamic layer 0 and layer 1, arrangement (b) of Figure 21 has a single SD Servient on top. This Servient consumes all SD Servients of layer 1.

Since it was possible to choose between a verified or regular commissioning phase of the SD Servient, this work created two measurement scripts. The behavior of the measurement script was equal except of the subprocess calls on SD Servient creation. Whilst verified SD Servients ran the `testSystem` method of the Testbench, regular SD Servient implementations checked for a connection to each system component without verification. One important notice was that `testSystem` calls did not check if duplicate verification of components happened. In (b) arrangements with multiple SD Servients, layer 1 SDs as well as the layer 2 SDs verified TDs of layer 0. This was due to the recursive collection of all system links of the `testSystem` method. Apart from that, the measurement scripts set both repetition and scenario numbers of the `testSystem` configuration to 1.

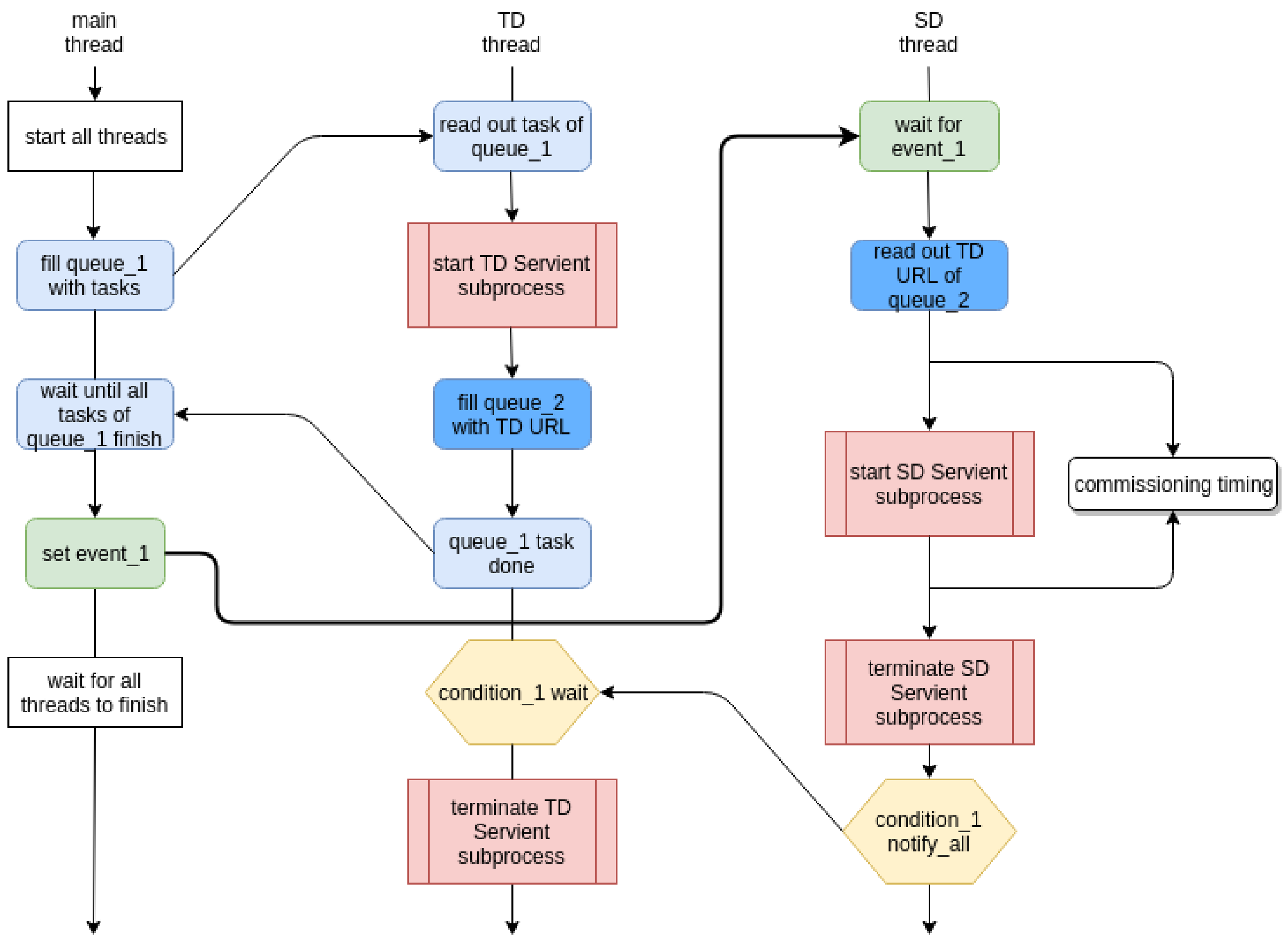


Figure 22 Thread model of evaluation measurement script for single System Description Servient arrangement

With knowledge of possible SD and TD arrangements and measurement script options, it is possible to address the thread model of Figure 22 and pseudocode of Listings 10 and 11 of the measurement script.

In Figure 22, the main thread starts all TD and SD threads. The quantity of TD and SD threads depended on input arguments of the measurement script. When looking at the pseudocode, lines 13 to 17 and lines 19 to 22 of Listing 10 initialize and start all child threads. This measure executed the `tempServient` and `sysServient` functions of Listing 11. But before executing child threads, Listing 10 parses arguments that determine the number of child threads in line 4 and 5. Thread model data structures such as events, queues, and conditions are initialized in the lines 7 to 11 in Listing 10. Figure 22 colors processing steps of events, queues, and conditions in green, blue, yellow respectively. Input to child threads of Listing 10 were the `tempServient` and `sysServient` functions of Listing 11 together with thread specific data structures. The event, condition and queue data structures allowed thread to thread communication and synchronization.

Listing 10: Scalability testing main thread

```
1  import sys,subprocess,threading,time,queue
2  from threading import Thread
3
4  args = sys.argv[1:]
5  anzahlTempServients = args[0]
6
7  # init data structures
8  c1 = threading.Condition()
9  e1 = threading.Event()
10 q1 = queue.Queue()
11 q2 = queue.Queue()
12
13 # init threads
14 sysThread = Thread(sysServient,c1,e1 q2)
15 tThreads = []
16 for i in range(anzahlTempServients):
17     tThreads.append(Thread(tempServient,c1,q1,q2))
18
19 # start threads
20 sysThread.start()
21 for x in tThreads:
22     x.start()
23
24 # feed tThreads
25 for x in range(anzahlTempServients):
26     q.put(x)
27 q.join
28 e1.set()
29
30 # end threads
31 sysThread.join()
32 for x in tThreads:
33     x.join()
```

Listing 11: Scalability testing Servient threads

```
# starts temperature Servient subprocess
def tempServient(c1, q1, q2):
    while True:
        # get queue value
        i = q1.get()
        # starts subprocess
        subprocess("tempServient.js", port+i)

        # insert in second queue
        q2.put("link_of_exposed_Servient")
        # marks queue task as done
        q1.done_task()

        with c1:
            # waits for notification
            c1.wait()
            # terminates subprocess
            subprocess.terminate()
            return

# starts system Servient subprocess
def sysServient(c1, e1, q2):
    # wait for event set
    e1.wait()
    links = []
    while True:
        # read out queue
        if not q2.empty():
            links.append(q2.get())

        # timed subprocess start
        start = time.time()
        subprocess("sysServient.js", links)
        end = time.time()
        print(end – start)

        subprocess.terminate()
        with c1:
            # notifies condition in threads
            c1.notifyAll()
            return
```

After thread initiations and start ups, the `sysServient` thread waited for the first event. The `sysServient` thread of Listing 11 represents the SD thread of Figure 22. The `tempServient` thread of Listing 11 which represents the TD thread of Figure 22 waited for elements from the first queue. Now, the main thread inserted increasing numbers which use the TD quantity number as limit into the first queue. Afterwards, the main thread waited until all tasks of this queue complete. Listing 10 indicates these steps in lines 24 to 27. Figure 22 marks these steps with its first blue queue field of the main thread.

Now, the TD thread of Figure 22 as well as function `tempServient` of Listing 11 launched a TD Servient subprocess. The increasing numbers of the queue served for spawning TD Servients with different ports. Parsing console logs of booting Servients ensured the detection of successful Servient launches. The URLs of the successfully exposed TDs expanded the second queue thereafter. Line 10 of Listing 11 shows this step. Likewise, the TD thread of Figure 22 indicates the step with a darker blue queue processing step. Next, TD Servient child threads marked the task of the first queue object as done and waited for notification of the first condition.

After completion of all queue tasks, processing in the main thread continued. This completion of tasks meant successful activation of all TD Servients. The next step which set the first event object caused the SD thread to continue processing. Line 28 of Listing 10 as well as the thicker arrow of Figure 22 showcase the step. The `sysServient` function of Listing 11 read all exposed URLs of the second queue. These links set the required `featureList` links for the regular as well as verified SD Servient. Both System Servient implementation used the provided links during their commissioning phase. After receiving the links from the second queue, the `sysServient` function started the SD Servient in another subprocess which can be seen in line 33 of Listing 11. As the script measured commissioning times of Servients, it measured and printed the time difference of before and after the subprocess call. Lines 31 to 35 of Listing 11 mark the timing implementation. After the subprocess call which started up the SD Servient with or without the `testSystem` method, the system Servient thread terminated the running subprocess. Termination happened after the detection of a console log which indicated successful commissioning of the SD Servient. Finally and before returning, line 40 of Listing 11 shows that the SD thread uses the first condition to notify all TD threads. This notification resolved waiting of TD threads. Eventually, line 18 of Listing 11 indicates termination TD Servients. Lines 30 to 33 of the main thread terminate all threads by calling the join method.

These consecutive steps of Listing 11 are also displayed in the SD thread of Figure 22. Here, the red subprocess fields mark SD Servient start up and termination. The shadowed timing tag indicates the start and end of the commissioning time. The yellow condition field re-activates thread processing of TD child threads. Re-activation of all child threads let them complete. This causes the main thread of Figure 22 to complete as well.

As mentioned, the explained code pattern allowed to build the Servient arrangement (a) of Figure 21. For extending the code concept to allow Servient arrangements with three layers, the implementation leveraged lists for managing SD Servient threads. For initializing threads of in lists, lists elements of mentioned data structures ensured the input to thread initialization functions. Servients started up layer by layer. This meant, that the successful activation of the Servients of layer one triggered the start up of SD Servients of layer two. One event, one condition, and two queue objects were necessary to ensure consecutive Servient start up. Now with the requirement of three layers of Servients, a second set of thread communication compatible data structures was necessary. With the second set of data structures, a list of system Servient threads, and another third layer Servient which consumed layer one Servients, this work implemented the three layer arrangement (b) of Servients shown in Figure 21.

6.2. Evaluation of System Description scalability

With the help of the measurement script from section 6.1, it was possible to collect the data of Tables 1, 2, and 3. Executing the script with different input numbers allowed to create versatile timing results. The timing always referred to SD Servient commissioning times. Tables provide data of virtually composed Servient mashups whereas another table Servient commissioning on Raspberry Pi devices measures. The following subsections report and comment all gathered timing results. Moreover, they differentiate between two methods of growing a system. The first subsection keeps constant SD arrangement and increases layer 0 TD components of the system. The second subsection keeps a constant relation between TDs per SD.

6.2.1. Scalability evaluation of virtually composed mashups

Table 1 provides virtual commissioning times in seconds of SD Servients of virtually composed mashups. The table provides numbers of SD Servients in the left two columns. The other columns have varying numbers of TD Servients. All tables of the evaluation chapter use the second row of the left two columns to mark whether SD Servients use the Testbench `testSystem` method during commissioning. A result with no input under the `verified` or `not-verified` columns uses the respective commissioning option during the measurement.

Table 1 uses constant SD Servient numbers per row. These numbers vary between 1, 3, and 6. The SD arrangement thereby varies between arrangements (a) and (b) of Figure 21. In the case of one SD Servient, the measurement script composes the arrangement (a). In the case of multiple SD Servients, the measurement script uses arrangement (b) where one SD Servient serves as a layer 2 Servient.

Table 1 Virtual commissioning times in seconds of constant System Description Servients with different numbers of Thing Description Servients

Number of SD Servients		Number of TD Servients									
not-verified	**verified**	**1**	**5**	**10**	**20**	**40**	**50**	**100**	**200**	**400**	**700**
1		0.13	0.14	0.16	0.19	0.29	0.32	0.53	0.91	1.93	3.64
3		0.28	0.30	0.35	0.35	0.46	0.54	0.79	1.29	2.62	3.79
6		-	-	-	-	-	0.77	1.00	1.53	2.68	3.50
	1	0.23	0.25	0.32	0.37	0.51	0.55	0.90	1.56	3.32	6.64
	3	0.49	0.58	0.67	0.79	0.93	1.07	1.59	2.60	5.58	n/a
	6	-	-	-	-	-	0.86	1.95	2.98	5.80	n/a

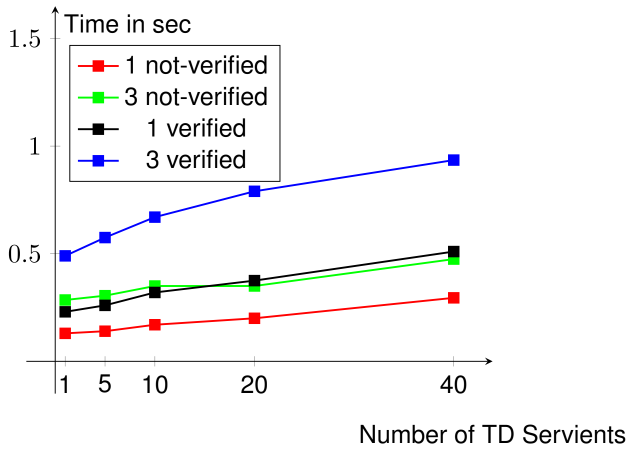


Figure 23 Line plot of commissioning times of constant System Description Servient numbers and increasing Thing Description Servient numbers from 1 to 40

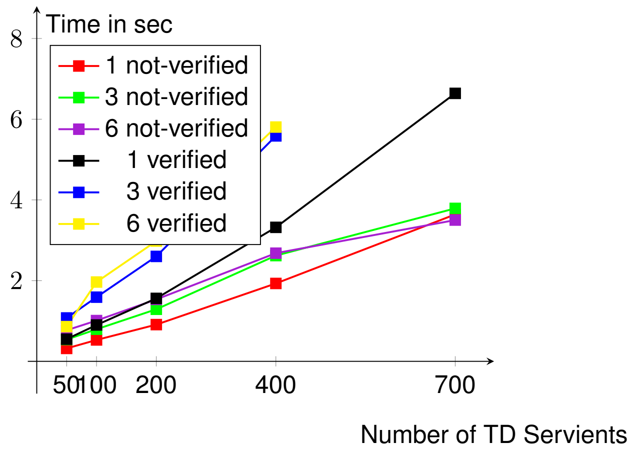


Figure 24 Line plot of commissioning times of constant System Description Servient numbers and increasing Thing Description Servient numbers from 50 to 700

By taking row three with 6 SD Servients as an example, Table 1 uses a virtual Servient composition with five SD Servients in layer 1 and one SD Servient in layer 2. Thereby, the total number of SD Servients is 6. For mashups with a TD number of 50 or above, Table 1 shows commissioning times with additional mashup compositions with 6 SD Servients. Numbers of TD Servients of the right section of columns of Table 1 reach from 1 to 700.

Figures 23 and 24 illustrate the data of Table 1. Figures 23 provides different numbers of verified and non-verified SD Servients which control an increasing number of TD Servients. The number of SD Servients in the legend is either 1 or 3 and the numbers of TD Servients on the x-axis increments from 1 to 40. The y-axis represents virtual SD Servient commissioning times in seconds. Figure 24 has the same layout as Figures 23. The differences are that numbers of TD Servients reach from 50 to 700 and the legend moreover displays an additional number of 6 SD Servients per tested mashup composition.

Before reporting and commenting details of data of Table 1 with the help of Figures 23 and 24, it is helpful to name general observations and trends of the data. All lines of Figures 23 and 24 show approximately linear increase. The reason why the blue and black lines reach higher data points compared to the red and green lines is due to the verification procedure during commissioning. Irregularities within data points to a general inaccuracy of measurements. By executing the same virtual mashup composition multiple times, measured commissioning times varied always. As a matter of fact, all provided values of the tables represent average values of two equal measurement samples. Missing values of the last column of Table 1 explain the missing data points of the blue and yellow line of Figure 24. It was not possible to collect timing results for these Servient arrangements due to limited computational capacity.

Figure 23 which plots times of smaller mashup sizes reveals a secondary trend. The times of single SD mashups lay closer together compared to the times of the mashup with three SDs. An explanation for this behavior is the verification difference. SDs with commissioning verification perform redundant TD testing. This is due to the fact that every SD Servient run the `testSystem` method. Layer 2 SD Servients also test layer 0 TDs which had been tested by layer 1 SD Servients. The redundant verification expresses itself in the line charts of both Figures 23 and 24.

A major trend of the data of Figure 24 which has larger mashup sizes indicates strongly increasing commissioning times of mashups with verified SDs. In comparison, commissioning times of mashups with non-verifying SD Servients grow towards linear. Interestingly, the non-verified line of Figure 24 which represents a mashup with 6 SDs has the last data point underneath both other non-verified lines. This means faster system commissioning with more SDs in the system. One remark is that the component size of this data point implies the largest virtually measurable mashup.

Stronger increase of times of verified mashups in Figure 24 clarify even more when comparing the larger verified mashups with data of the smaller mashups. Smaller verified mashups already show indications of longer commissioning times. As explained this is also due to redundant verification tests. Larger systems have drastically more components, this means that the redundancy of verification tests grows even more. The reason for this is that the `testSystem` method takes longer for testing more TDs. Redundancy leads towards exponentially growing commissioning times because longer testing procedures repeat.

To sum up first insights after inspecting Table 1 and Figures 23 and 24, mashups with multiple SDs scale better if the component size of the mashup is high enough. This outcome happens in the verified as well as non verified SD Servient case. Figure 24 proves this trend with slightly growing scopes of mashups with more SDs. The yellow line of Figure 24 shows this fact within the last value domain of the horizontal axis. Likewise non-verified systems with lines such as the red line of Figure 24 have a higher scope compared with the green and purple line which indicate mashups with multiple SDs.

6.2.2. Scalability evaluation of Raspberry Pi mashups

Table 2 Commissioning times in seconds of Raspberry Pi System Description Servients with different numbers of Raspberry Pi Thing Description Servients

Number of SD Servients		Number of TD Servients		
not-verified	**verified**	**1**	**2**	**3**
1		0.46	0.48	0.55
	1	1.36	1.48	1.58

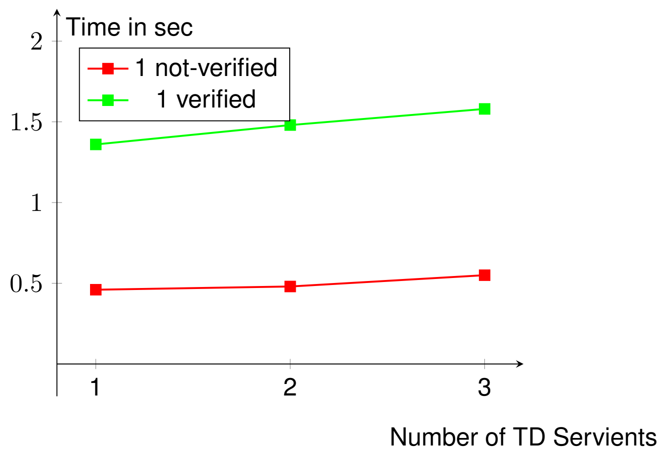


Figure 25 Line plot of commissioning times of Raspberry Pi System Description Servient and increasing Raspberry Pi Thing Description Servient numbers from 1 to 3

Table 2 follows the exact same design of Table 1. As well as Table 1, Figure 25 which plots the Servient commissioning times complements the data of Table 2. As there were three Raspberry Pi devices available, the mashup of Servients grew to a size of four Servients. One Servient thereby exposed the SD.

When looking at differences of virtual versus real mashup implementations, there is the number of SDs which stays at 1 in the case of the Raspberry Pi mashup. Clearly, this fact is due to the small mashup size. Secondly and by inspecting the line chart of the data of Figure 25, SD Servient commissioning takes about twice as long compared to the virtual Servient commissioning. Obvious reasons for this behavior are longer delays caused by communication between TDs. Moreover Raspberry Pi devices with 1 GB RAM have less computational capacity compared to the 16 GB RAM Macbook Pro which ran virtual mashup tests. Data points of the Raspberry Pi Servients take three times longer when comparing non-verified with verified system commissioning times. Data points of virtual mashups of Figure 23 approximately double when looking at this same comparison. The commissioning times of SD Servients of the Raspberry Pi mashup increase with the increasing number of TD Servients. The minimal time increase follows the behavior of the data gained through collecting virtual mashups commissioning times. Figure 25 shows no unexpected outcomes and aligns to the findings of Table 1.

6.2.3. Scalability evaluation of constantly increasing system sizes

The following scalability testing approach increases the system size while keeping the amount of TDs per SD. There is no comparison of this scalability approach to a real mashup implementation. The three Raspberry Pi devices show that it is possible to apply the proposed SD in a real scenario.

Table 3 shows commissioning times of virtual SDs Servient mashups. It uses the left two columns to indicate numbers which stay constant during system growth. These numbers of the left two columns mark the numbers of TD Servients per SD Servient. The second and forth row of the left two columns of Table 3 indicate whether SD Servients perform verified commissioning. The respective right columns of these rows determine the system mashup Servient arrangement. The smallest SD mashup setup of Table 3 uses 50 TDs and one SD. The next size with 100 TDs and two SDs doubles the first size. It moreover uses a third SD Servient on top which manages the two layer 1 SDs. The next size composes of four layer 1 SDs with each managing 50 TDs. This means total numbers of 200 TDs and 5 SDs. Servient arrangement numbers of row four of Table 3 follow this same principle of system growth.

Figure 26 visualizes the data of the Table 3. Again, all values of Table 3 and Figure 26 are average values of two commissioning time samples. An irregularity of the line chart in Figure 26 sets the blue line which stops after two data points. The reason for this was that virtually composed mashups exceeded the computational capacity of the computer used for measuring commissioning times.

Table 3 Virtual commissioning times in seconds of constantly growing Thing Description Servients per System Description Servient

Number of TD Servients per SD Servient		Number of TD-SD Servients			
not-verified	**verified**	**50-1**	**100-2**	**200-4**	**400-8**
50		0.32	0.79	1.36	2.64
	50	0.56	1.62	2.86	6.02
not-verified	**verified**	**50-5**	**100-10**	**200-20**	**400-40**
10		0.63	1.13	2.19	4.34
	10	1.34	2.58	n/a	n/a

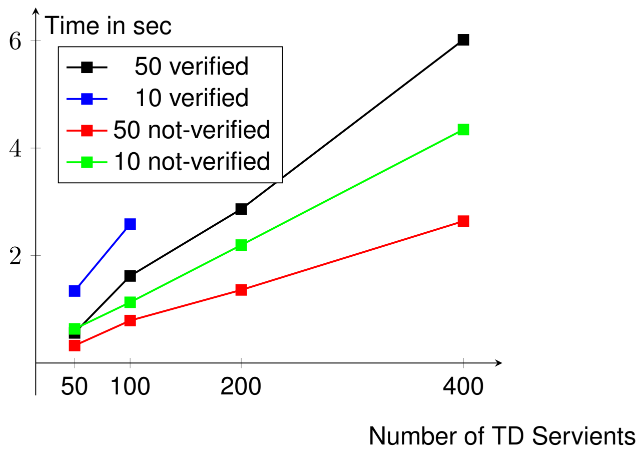


Figure 26 Line plot of virtually measured commissioning times of constantly growing System Description mashup sizes

A trend of the data of Figure 26 is approximately linear growth. Again, this behavior of increasing commissioning times is due to the increasing size of the mashup components. The black line has a larger scope compared to the red and green lines. The reason therefore is that the black line represents mashups with verified SD Servients. As interactions between TDs and SDs increase with verification checks, the black line grows faster with an increasing number of SDs. In contrast to Figures 23 and 24, the scope of the first two data points of the black and red lines of Figure 26 are higher compared to successive scopes of the data. An explanation for this behavior is that these mashups start with a single SD but continue with an additional layer 2 SD Servient. All other Servient mashups have constant SD increase.

To sum up, it is possible to say that constantly growing SD mashups scale approximately linear. Even though the black line of Figure 26 has a higher scope compared to non-verified system commissioning times, it is still possible to say that constant growing mashups scale. This statement counts for mashup sizes investigated in this work. When looking at all outcomes together, it is possible to suggest to switch to the use of mashups with multiple SDs when the number of TDs grows over 400. Afterwards, larger mashups with constant growing SD to TD ratios continue to scale well. The verification approach during system commissioning ensures adaptive capabilities of the SD already during start up. The verification procedure of the `testSystem` method requires more interactions compared to constant interval fetches of system information by a controlling unit. Therefore it is possible to conclude that runtime system adaptation scales at worst case as well as verified mashups used for measuring commissioning times.

6.3. Discussion

The outcomes of the evaluation suggest the usage of a SD which extends TDs with minimal system specific features. The implementation close to the WoT stack allows the SD interactions with other Things. The integration of System information in TDs strengthens interoperability between system components and the controlling component of the system. This factor contributes well to the results of the scalability testing results which measured linear growing commissioning times with increasing system sizes.

The verification procedure used during system commissioning enables adaptive SD behavior. The system runtime adaptation technique with the use of a controlling unit that uses interval based component checks requires less verification interactions inside the system compared to the commissioning verification. The interval system runtime adaptation approach with one controlling unit moreover combines system functionality tasks with component monitoring. Therefore the verification procedure of the commissioning phase of systems with multiple SD leaves space for optimization. The best solution for the measurement script is to use a system verification Servient in the highest possible hierarchy. The `testSystem` method has the capability of recursive link collection of system components. With only one `testSystem` call from the top hierarchical layer would not produce redundant checks and fasten the commissioning process. In systems with more complex application scenarios than the simple average temperature, this measure would save even more time. Saving time during commissioning simultaneously means speed up of system runtime adaptation.On the other side, systems with SD subsystems never know whether they are part of a management unit which interacts on top themselves. Therefore it is also important to make sure that all sub components in their own system work. The measurement script builds up virtual mashup compositions where SD Servients verify underlying components one after each other. The minimal time delay during virtual consecutive system checks might change in larger and more complex real life scenarios of mashup compositions. Consecutive component checks will moreover ensure that the SD adapts and exposes itself with a higher probability of correctness.

It is important to note that the amount of interactions of the verification procedure affect commissioning times. The evaluation graphs indicate this behavior even though the system application is very simple. It is assumable that a more complex application scenario with for example complex inter system communication protocols would negatively impact the scalability of the overall mashup. It is not safe to say that the implemented controlling unit in combination with SDs performs well in cases of complex large systems.

The outcomes of this work align with statements of other research. Similarly to this work, the work of Kovatsch et al. [28] evaluates the increasing number of services with regard to time. Instead of measuring the commissioning times, the work [28] measures reasoning times of a reasoner which determines automated system execution plans. This related work states that they neglect communication times inside their systems during automated service composition. They moreover state that those times can have a significant role in constrained environments. Even when mashups are statically configured they experience overhead when invoking service interactions. This work detects the overhead of interactions in the evaluation. Here, the verification procedure with affects commissioning times even though the composition is statically composed. The evaluation of the work of Kovatsch et al. [28] proves approximately linear scalability in their service composition approach. Their stepwise increase in the complexity of the execution plan expresses with smaller irregularities in timing plots. Similarly behave the figures of this work with increasing SDs and thereby complexity. The larger starting scope in Figure 26 is due to an additional SD. Therefore increasing complexity also expresses in evaluation graphs of this work. As a last fact, the work of Kovatsch et al. [28] assumes longer reasoning times with Raspberry Pi types of devices. The evaluation with plot 25 alignes with this assumption as well.

Overall it is possible to say that the outcomes of this work align with related research. This counts for the investigated system sizes. Growing complexity and system size scale approximately linear in the provided evaluation which measures statically composed commissioning times of SD dependent mashups with adaptive capabilities.

7. Limitations

The limitation chapter of this work explains occurring problems which came up during Testbench and SD implementation. Especially the evaluation part unveils limitations which simultaneously indicate future work suggestions. The chapter structures itself with two comprehensive lists about Testbench and SD limitations. Each section lists and further explains details of the point of the listing.

7.1. Testbench

It is important to note that most of the mentioned limitations originated from feedback of Bundang plugfest mashup developers which used the Testbench. The following points of the listing refer to Testbench specific limitations only:

- Missing security protocol implementation

  The first limitation addresses missing security protocol compatibility. The implemented Testbench exposed itself through HTTP and CoAP protocols. Without certificate management, no interactions based on HTTPS and CoAPS were possible. This limitation has the highest priority since HTTPS and CoAPS set state of the art security standards. Plus, many industrial devices tested in plugfest mashups fully depend on these security protocols.

- Tedious installation and set up procedure

  The next limitation refers to the installation procedure. Even though there is a documentation of the installation process, there are to many steps and navigation commands necessary in the console to set the Testbench project up. This limits usability. An shell or installation script could facilitate the problem drastically.

- Missing state behavior

  Another limitation comes with the Testbench as a service design. People that try to interact with one running Testbench have no knowledge about the state of the Testbench. The Testbench does not expose whether it is in a busy, idle, or broken state. This problem limits usability in a way that two mashup developers might send the TD of the Thing they like to test simultaneously. Consecutive write requests force the current Testbench implementation to overwrite the previous received values. This problem also refers to other interaction possibilities such as setting the configurations of the test. Following classical state machine system states and exposing this state as a property is a potential solution for this problem.

- No event type interaction testing

  One of the last points addresses the missing test method for the event type of TDs. This method was missing before the start of this work due to missing W3C specifications about the interaction type. Even though specifications about the event interaction type have greater detail now, it was not possible to finish a fully covering testing method for the TD event interaction type. With that said, the event interaction type testability remains an open issue which is marked in the Testbench documentation as well.

- Bundang plugfest shortly after new Thing Description specification draft

  Since the W3C published an updated TD working draft shortly before the Bundang plugfest, it was not possible to implement additional add-ons on the Testbench due to time constraints. Without time constraints, it would have been possible to develop an verbose event testing method.

7.2. System Description

Limitations of the SD are structured in the same way as the limitations about the Testbench. A bullet point list with explaining paragraphs summarizes all limitations.

- Evaluation data mostly of virtual mashups

  The scalability evaluation reveals limitations of this work that affect the SD. Since the evaluation of this work heavily utilizes virtual system compositions, there is only limited knowledge of how the SD approach scales with more than three real world devices. Future investigations on this proposed SD approach have to further research real world applicability. Upcoming plugfests could contribute to this issue perfectly.

- Controlling unit implemented on System Description Servient

  Next, this work implemented the controlling unit on the same Servient which exposed the SD. As this design simplified and satisfied the implementation and demands of mashups of this work, it limited modularity of the SD approach. Since flexibility is an important factor in WoT concepts, this limitation might require different implementations in other mashup scenarios. A modular component with controlling capabilities allows easier replication in distributed systems for example.

- Virtual mashups tested in a single compatible environment

  One remark on the testing environment is the fact that the measurement script build up system components using one environment for system commissioning. Real life scenarios such as the Raspberry Pi mashup require complete environments with all dependencies to work. This work does not consider fully automated environment builds which would improve real world scalability simulation even more. Container environments and container orchestration could provide an opportunity to further optimize scalability testing of closer real world commissioning.

- Biased evaluation data due to testing computer

- Verification redundancy in evaluation mashup concept

  These points address biased measurement data and verification redundancy during commissioning of SD Servients. The computer used for measuring commissioning times delivered differing measures. It was not possible to investigate the reasons for this behavior within the scope of this work. Furthermore computational expensive measurements with larger Servient arrangements produced stronger varying times. The recursive `testSystem` method of the verification procedure in combination with the design of Servient arrangements for commissioning measurements limited the adaptive capabilities of the SD with regard to time. Redundancy happened due to repetitive verification checks on same TDs. This limitation will affect larger and more complex systems even more.

- System component needs at least computational capacity to run WoT Servient

  It is the fact that every device in a SD controlled system requires at least computational capacity to run a WoT Servient implementation. This problem limits the applicability of SDs as not all devices are capable of running WoT Servients.

- System Description design particularly addresses adaptive capabilities and scalability

Lastly, it is necessary to say that the SD was designed with particular emphasis on adaptive and scalable capabilities. SD applications of this work did not show limiting behavior with regard to system flexibility, security, or compatibility. Nevertheless, the implemented SD application utilized a very simple scenario. It is assumable that average WoT mashups deal with higher complexity. They thereby potentially face limitations on other system properties with the application of this SD approach.

8. Related Works

A lot of effort and progress has been made within the WoT community to tackle problems of heterogeneity and scalability, security and privacy, search and discovery, and ambient intelligence of WoT devices. There have been IoT platforms and ecosystems which allow to create environments and systems of compatible devices. Most implemented systems promote their individual solutions which only enable individual device interoperation. They thereby tackle problems of versatile data formats, device interfaces, and protocols. As a result and in order to implement the same functionality in different platforms, developers and customers have to re-implement the same functionality from scratch. This leads to slow adoption of existing solutions and slows progress of IoT platforms and ecosystems [24]. One direction to solve interoperability and scalability problems which derive from high IoT device versatility are automated system composition techniques. Automated goal driven configurations for system compositions of smart environments show facilitation in building WoT networks and with that device management systems. Here, configuration automation provides system independence of ambient smart environments [44]. After explaining the related work of Mayer et al. [44], this chapter introduces related research about TDs which solve system specific interoperability, scalability, and protocol problems.

8.1. System runtime adaptation and scalability

As already stated in the beginning of this chapter, the work of Mayer et al. [44] provides a possible solution for the interoperability problem of ambient environments such as hotel rooms, public places, and office environments. An advantage of the work of this paper is that the systems have high flexibility, scalability, and adaptive capabilities with regard to dynamic computing environments and device unavailability. For achieving these system capabilities, their work utilizes a Web based search infrastructure based on HTTP and CoAP for the discovery of services and their semantic descriptions. The semantic descriptions are based on RESTdesc [63] which is a light weight semantic format that allows automatic compositions based on provided functionality. A semantic reasoner module takes this capability of automatic composition for composing systems based on a user defined goal. It is not specified where the hosting of the semantic reasoner happens. But due to the avoidance of security implications and efficient communication delays, the reasoner runs on a local instance. The semantic reasoner receives the user defined goal from a client application of the user. This design and the advantage of clients to obtain execution plans of their queries very rapidly allows maximal adaptation. The reason therefore is that clients may re-query the reasoner during execution of a goal through a service.

The evaluation of the work of Mayer et al. [44] investigates reasoning times for a growing number of services with descriptions. The evaluation originates from the work of Kovatsch et al. [28] which the evaluation Chapter 6 uses for discussing outcomes. The major trend and outcome of the evaluation of the work of Mayer et al. [44] is the approximately linear growth of reasoning times for growing service numbers. The paper of Kovatsch et al. [28] moreover unveils additional information about adaptive methodologies of the reasoner. For monitoring services, the reasoner caches all semantic descriptions of the services locally. To do so, it periodically checks a resource directory which for updating information about discovered services. The reasoner adds or removes descriptions from the cache depending on the information of the resource directory. This process ensures monitoring and adaptivity of services for automatically composed systems of the works of Kovatsch et al. [28] and Mayer et al. [44].

Compared to the work of Mayer et al. [44], the first similarity of this work uses a controlling unit embedded into the SD Servient implementation. This measure resembles the methodology of local hosting of the reasoner of the paper of Mayer et al. [44] and brings equal advantages. A main difference is that the SD build upon the TD standardization. RESTdesc is no standardized description. Moreover an event interaction type cannot be described by RESTdesc. Compared to the RESTdesc description of the work of Mayer et al. [44], the TDs and SDs of this work are human readable and writable.

Regarding the evaluation, a difference between the mentioned works and this work is that this work evaluates SD Servient commissioning times instead of reasoning times. Similarly the evaluations test scalability and grow system sizes with additional TDs and services respectively. An important difference between these works is the fact that the work of Mayer et al. [44] apply richer diversity of services and high complexity system compositions. Whereas this work focuses on a first simple implementation of a system which includes one or multiple SDs.

Very similar to the monitoring implementation of the work of Kovatsch et al. [28], this work uses periodic TD fetches for enabling monitoring capabilities in the system. The monitoring capabilities again allow adaptive system behavior of the SD of the system. The difference between both adaptation approaches is that the interval fetch of this work comes from an extra system unit. The controlling unit gathers system adaptation tasks into one module. Generally it is possible to say that these two works implement very basic adaptation concepts. Today's analytical possibilities allow predictions of runtime failures and guarantee more reliable system adaptation processes [64]. But for the type of systems investigated in these related works, basic adaptive and monitoring methodologies such as repetitive requests [21] satisfy research questions.

8.2. Thing Description

The work of Charpenay et al. [65] investigates and deals with the development of TDs concept proposed by the W3C working group. Their research emphasizes the alignment of TD concepts with other IoT vocabulary. In other words, the work formally defines the two concepts of TD and interaction in Semantic Web vocabulary. Their proposal of TD and interaction concepts in semantics is based on the Identifier, Resource, Entity (IRE) model. This model is one of the main data structures of semantic notations. To be able to analyze alignment of of their proposed structure with existing IoT vocabulary, their work uses vocabulary graphs and the criteria of complementation with Linked Open Vocabularies (LOV). LOV are published data repositories with the purpose to make data accessible through links and queries. Secondly the work captures and introduces a common understanding of the TD and interaction core WoT resources.

The relation of the work of Charpenay et al. [65] comes with the overall methodology. The SD proposal of this work depends on the study of existing TD vocabulary. In a similar way but instead of investigating WoT interactions and WoT TDs with regard to semantics, this work investigates WoT TDs and stack properties with regard to system compositions capabilities. Instead of using different semantic stack vocabulary such as the IRE structure as the work of Charpenay et al. [65], this work sticks to the WoT stack for creating a SD proposal. A similarity of this work with regard to the work of Charpenay et al. [65] is that it develops a general understanding of WoT concepts but in a different domain. The field of this work deals with scalable and adaptive system solutions. In comparison, the work of Charpenay et al. [65] investigates alignments to IoT vocabulary.

The related work of Thuluva et al. [16] investigates IoT system composition. Systems are abstracted with the term recipes. IoT device functionalities serve as offerings or ingredients for recipes. The work moreover implements a recipe cooker which lets users choose IoT offerings as ingredients. The recipe cooker automatically composes applications and matches interfaces and protocols of IoT offerings. The ontological model for describing offerings called Offering Description (OD) model is based on TDs. The OD model extends TDs with JSON keys for required functional concepts of the model. In their case, TD extensions add marketplace specific and discovery helpful information to the OD.

Similarly to the OD of the work of Thuluva et al. [16], the newly proposed SD of this work is based on TDs. System specific information and JSON keys help controlling units to detect necessary information by parsing the SD. SD exposed information allows consuming services to perform system specific tasks. The Testbench `testSystem` method for instance only works with the help of the component links in the SD. Analogously to the term recipe, this work uses a SD for describing system characteristics. A recipe combines the interplay of ingredient capabilities to form a new dish. In a similar way, the SD is able to combine component specific functionalities for expressing a new application.

9. Conclusion

This work investigates TD standardization from the W3C working group. It first contributes to the development of the TD testing tool called Testbench. Thereby improvements update and transform the Testbench into a REST service. Afterwards and with knowledge about TDs, this work proposes a WoT stack compatible SD. SD design and implementation focus on scalability and adaptivity capabilities of SD applications. The final scalability evaluation of adaptive SD mashups proves approximately linear scalability.

The use of the updated Testbench during the W3C Bundang plugfest 2018 justifies the first research question of this work. Thereby, documentation and REST service features of the Testbench advances usability. A new SD testing method developed for verified SD Servient commissioning further extends the Testbench. Linear scalability and adaptive commissioning and runtime of SD applications show that adaptive SD development is possible. The outcomes of this work moreover show that mashups composed of multiple SDs scale better with a high amount of system services compared to single SD mashups.

With respect to the overall study area, it is possible to say that this work extends the existing TD concepts with regard to system behavior. Therefore it proposes a SD which has a TD basis and additional new properties for system compatibility. This implementation of a SD reveals that the TD concept provides flexibility with regard to new extensions. Moreover this work shows that it is possible to apply TDs as well as SDs in WoT mashups. Next to the IoT interoperability problem, the SD of this work additionally solves issues of system adaptivity and scalability. Therefore, the applied SD design utilizes a mixture of distributed and central architectural components as well as modular controlling units. To sum up, this work provides an option to build scalable, adaptive, and extendable WoT mashups with the help of SDs. The only requirement of the SD approach is that all system components respond with regard to their TD.

9.1. Future Work

The following section suggest future research with regard to the updated Testbench and the introduced SD concept. It thereby builds upon outcomes and limitations of the evaluation 6 and limitation 7 chapters.

9.1.1. Testbench

Future work suggestions for the Testbench mostly rely on mashup developer feedback. Therefore it is necessary to add configuration options for protocol security settings to the configuration property of the Testbench. It is moreover recommendable to provide an installation script or shell script for setting up the Testbench Servient. Since the Testbench is a stateless service, the integration of a state machine property could tackle the problem of property overwriting due to simultaneous client interaction. Other improvement directions of the Testbench are the integration of the event type interaction testing method and development towards facilitated mashup integration. Facilitated mashup integration means new Testbench interactions and methods which simplify deployment of the Testbench in existing mashups. The SD testing method sets an example for development towards mashup integration since it increases Testbench functionality with regard to system characteristics.

9.1.2. System Description

Future work of this subsection refers to recommended updates for the SD and the controlling unit. The controlling unit of this work has basic monitoring capabilities which allow adaptation of the SD. For extending adaptive methodologies, the controlling unit could perform threshold checking during an interval loop. The event when a value exceeds its threshold could be used for adapting the SD. The paper of Taylor et al. [22] uses threshold detection and temporal relationships of events for triggering adaptation processes. To advance adaptation, the usage of predictive analytical models would allow to optimize decisions of the controlling unit. An intelligence gain could lead to failure prediction instead of failure adaptation of the SD.

The SD approach of this work divides its interactions into functional or controlling interaction categories. Nevertheless, it is possible to think of other interaction categories such as a security interaction category. When coming up with a new SD specific interaction for system characteristics, it is very important to think of security implications. The SD of this work exposes component URLs. This is not the best implementation with regard to security. Because an attacker immediately knows and is able to attack these system internal links. A method to disguise mandatory information for an controlling unit could be solved by providing an empty non-writable property with secure access authentication. In this case, the property handler function would return system information to authenticated clients only. Interaction types with regard to security requirements represent a candidate for an interaction category.

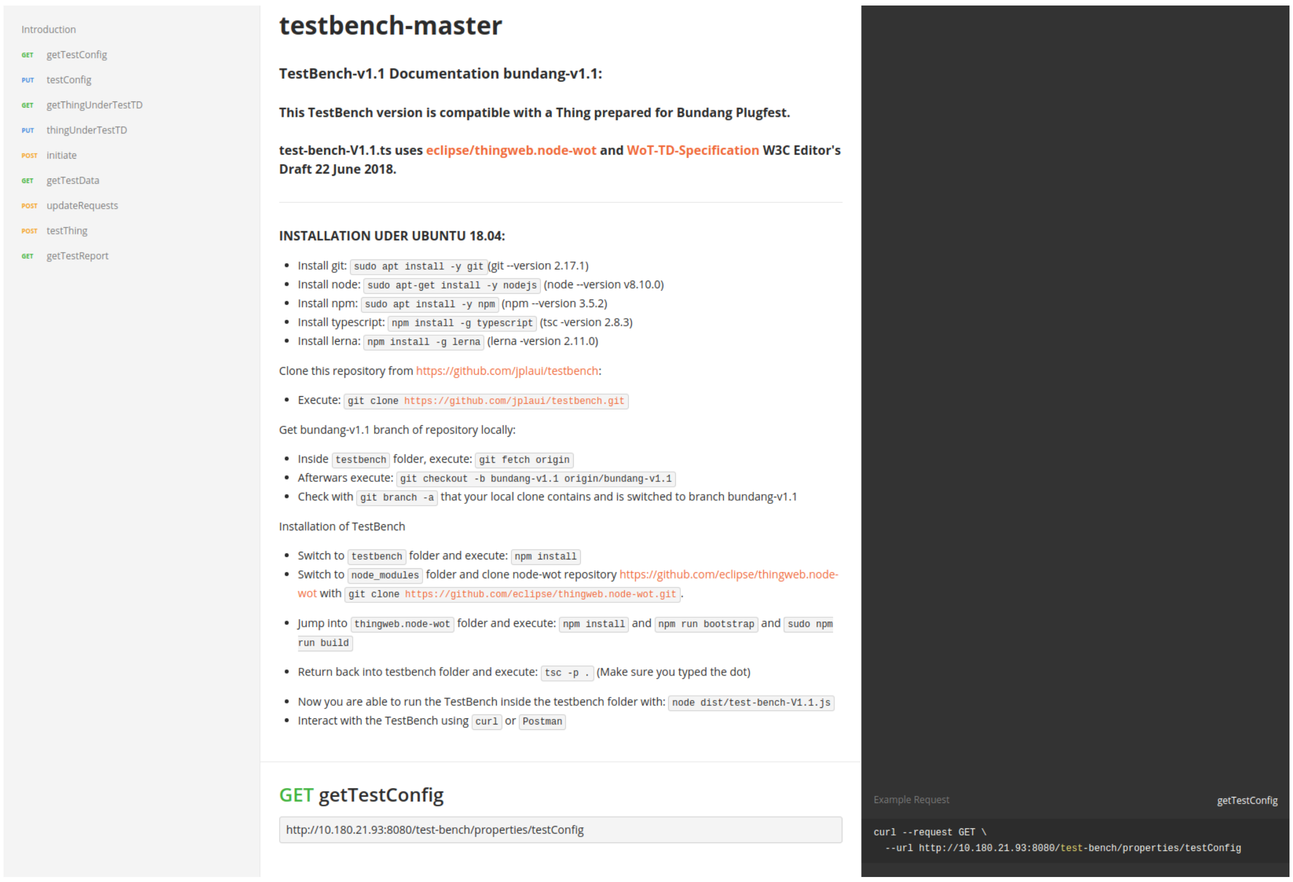
Other interesting extensions of SD interactions would be writable properties. An writable property of the system size for instance would require automatic component discovery, bootstrapping, and removal. As proposals for new interactions often depend on requirements of the application the mashup provides, it is rather recommendable to motivate usage of SD in more real life applications. The Raspberry Pi mashup for SD implementation set a very simple application scenario.

This work implements a very simple application scenario for testing the SD. As there are more complex systems available today [28], it would help to apply the SD in more complex mashups for clarifying pros and cons that come with SD usage. It is also thinkable to directly include adaptive logic of the controlling unit into a function that handles interaction calls. This would provide adaptive behavior of single interactions of the SD. But again, most efficient adoption of such an approach is determined by the application of the SD. For further promotion of the SD approach of this work, it is very recommendable to use the implementation of the SD in WoT plugfests.

Appendices

A. Appendix

A.1. Additional Figures



testbench-master

TestBench-v1.1 Documentation bundang-v1.1:

This TestBench version is compatible with a Thing prepared for Bundang Plugfest.

test-bench-V1.1.ts uses eclipse/thingweb.node-wot and WoT-TD-Specification W3C Editor's Draft 22 June 2018.

INSTALLATION UDER UBUNTU 18.04:

- Install git: `sudo apt install -y git`(git --version 2.17.1)
- Install node: `sudo apt-get install -y nodejs` (node --version v8.10.0)
- Install npm: `sudo apt install -y npm` (npm --version 3.5.2)
- Install typescript: `npm install -g typescript` (tsc -version 2.8.3)
- Install lerna: `npm install -g lerna` (lerna -version 2.11.0)

Clone this repository from https://github.com/jplaui/testbench:

- Execute: `git clone https://github.com/jplaui/testbench.git`

Get bundang-v1.1 branch of repository locally:

- Inside `testbench` folder, execute: `git fetch origin`
- Afterwars execute: `git checkout -b bundang-v1.1 origin/bundang-v1.1`
- Check with `git branch -a` that your local clone contains and is switched to branch bundang-v1.1

Installation of TestBench

- Switch to `testbench` folder and execute: `npm install`
- Switch to `node_modules` folder and clone node-wot repository https://github.com/eclipse/thingweb.node-wot with `git clone https://github.com/eclipse/thingweb.node-wot.git`.
- Jump into `thingweb.node-wot` folder and execute: `npm install` and `npm run bootstrap` and `sudo npm run build`
- Return back into testbench folder and execute: `tsc -p .` (Make sure you typed the dot)
- Now you are able to run the TestBench inside the testbench folder with: `node dist/test-bench-V1.1.js`
- Interact with the TestBench using `curl` or `Postman`

GET getTestConfig

http://10.180.21.93:8080/test-bench/properties/testConfig

Example Request getTestConfig

```
curl --request GET \
  --url http://10.180.21.93:8080/test-bench/properties/testConfig
```

Figure 27 Testbench Postman workspace with documentation and sample interaction requests

Bibliography

[1] The World Wide Web Consortium. Web of Things (WoT) Architecture. https://www.w3.org/TR/2017/WD-wot-architecture-20170914/, 2017. Online; accessed 4 June 2018.

[2] The World Wide Web Consortium. Web of Things (WoT) Thing Description April 2018. https://www.w3.org/TR/2018/WD-wot-thing-description-20180405/, 2017. Online; accessed 12 June 2018.

[3] Ege Korkan, Sebastian Käbisch, Matthias Kovatsch, and Sebastian Steinhorst. Sequential behavioral modeling for scalable iot devices and systems. In 2018 Forum on specification and Design Languages (FDL), in press.

[4] The World Wide Web Consortium. wot-scripting-api/README.md at master · w3c/wot-scripting-api. https://github.com/w3c/wot-scripting-api/blob/master/applications/thing-directory/README.md, 2018. Online; accessed 8 July 2018.

[5] Xiaohua Ge, Fuwen Yang, and Qing-Long Han. Distributed networked control systems: A brief overview. Information Sciences, 380:117–131, 2017.

[6] json-schema-faker. json-schema-faker/json-schema-faker: JSON-Schema + fake data generator. https://github.com/json-schema-faker/json-schema-faker. Online; accessed 27 August 2018.

[7] Ege Korkan. wot/testbench-architecture.pdf at master · w3c/wot · github. https://github.com/w3c/wot/blob/master/testing/TestBench-Architecture.pdf, 2018. Online; accessed 17 August 2018.

[8] Postdot Technologies. Postman | API Development Environment. https://www.getpostman.com/.

Online; accessed 24 August 2018.

[9] json-ld.org. JSON-LD - JSON for Linking Data. `https://json-ld.org/`, 2018. Online; accessed 24 September 2018.

[10] Felix Wortmann and Kristina Flüchter. Internet of things. Business & Information Systems Engineering, 57(3):221–224, 2015.

[11] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. IEEE Communications Surveys & Tutorials, 17(4):2347–2376, 2015.

[12] Mark Pesce. The web-wide world. In Proceedings of the 26th International Conference on World Wide Web, pages 3–3. International World Wide Web Conferences Steering Committee, 2017.

[13] Federica Paganelli, Stefano Turchi, and Dino Giuli. A web of things framework for restful applications and its experimentation in a smart city. IEEE Systems Journal, 10(4):1412–1423, 2016.

[14] The World Wide Web Consortium. Web of Things Working Group Charter. `https://www.w3.org/2016/12/wot-wg-2016.html`, 2016. Online; accessed 4 June 2018.

[15] Deze Zeng, Song Guo, and Zixue Cheng. The web of things: A survey. JCM, 6(6):424–438, 2011.

[16] Aparna Saisree Thuluva, Arne Bröring, Ganindu P Medagoda, Hettige Don, Darko Anicic, and Jan Seeger. Recipes for iot applications. In Proceedings of the Seventh International Conference on the Internet of Things, page 10. ACM, 2017.

[17] Leslie Sikos. Web standards: mastering HTML5, CSS3, and XML. Apress, 2014.

[18] Shadi Abou-Zahra, Judy Brewer, and Michael Cooper. Web standards to enable an accessible and inclusive internet of things (iot). In Proceedings of the 14th Web for All Conference on The Future of Accessible Work, page 9. ACM, 2017.

[19] Borja Ramis Ferrer, Sergii Iarovyi, Andrei Lobov, and José L Martinez Lastra. Potentials of web standards for automation control in manufacturing systems. In Modelling Symposium (EMS), 2015 IEEE European, pages 359–366. IEEE, 2015.

[20] Ramon Sanchez-Iborra and Maria-Dolores Cano. State of the art in lp-wan solutions for industrial iot services. Sensors, 16(5):708, 2016.

[21] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain, volume 15, 2009.

[22] Kerry Taylor, Colin Griffith, Laurent Lefort, Raj Gaire, Michael Compton, Tim Wark, David Lamb, Greg Falzon, and Mark Trotter. Farming the web of things. IEEE Intelligent Systems, 28(6):12–19, 2013.

[23] Soumya Kanti Datta. W3c web of things interest group face-to-face meeting at eurecom [future directions]. IEEE Consumer Electronics Magazine, 5(3):40–43, 2016.

[24] Arne Bröring, Stefan Schmid, Corina-Kim Schindhelm, Abdelmajid Khelil, Sebastian Kabisch, Denis Kramer, Danh Le Phuoc, Jelena Mitic, Darko Anicic, and Ernest Teniente López. Enabling iot ecosystems through platform interoperability. IEEE software, 34(1):54–61, 2017.

[25] Sareh Aghaei, Mohammad Ali Nematbakhsh, and Hadi Khosravi Farsani. Evolution of the world wide web: From web 1.0 to web 4.0. International Journal of Web & Semantic Technology, 3(1):1, 2012.

[26] Roy T Fielding and Richard N Taylor. Architectural styles and the design of network-based software architectures, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[27] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. From the internet of things to the web of things: Resource-oriented architecture and best practices. In Architecting the Internet of things, pages 97–129. Springer, 2011.

[28] Matthias Kovatsch, Yassin N Hassan, and Simon Mayer. Practical semantics for the internet of things: Physical states, device mashups, and open questions. In Internet of Things (IOT), 2015 5th International Conference on the, pages 54–61. IEEE, 2015.

[29] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the web of things (distributed node-red). In Proceedings of the 5th International Workshop on Web of Things, pages 34–39. ACM, 2014.

[30] The World Wide Web Consortium. JSON for Linking Data. https://www.w3.org/TR/2014/REC-json-ld-20140116/, 2014. Online; accessed 12 June 2018.

[31] M Duerst and M Suignard. Rfc 3987, internationalized resource identifiers (iris)[rfc]. https://tools.ietf.org/html/rfc3987, 2005.

[32] Francis Galiegue, Kris Zyp, et al. Json schema: Core definitions and terminology. Internet Engineering Task Force (IETF), 32, 2013.

[33] The World Wide Web Consortium. Web of Things (WoT) Protocol Binding Template. https://www.w3.org/TR/2018/NOTE-wot-binding-templates-20180405/, 2018. Online; accessed 04 July 2018.

[34] Ian Fette and Alexey Melnikov. The websocket protocol. Technical report, 2011.

[35] The World Wide Web Consortium. Web of Things (WoT) Security and Privacy Considerations. https://www.w3.org/TR/2017/NOTE-wot-security-20171214/, 2017. Online; accessed 10 July 2018.

[36] David Recordon, Dick Hardt, and E Hammer-Lahav. The oauth 2.0 authorization protocol. Network Working Group, 5849:1–47, 2011.

[37] Simon Duquennoy, Gilles Grimaud, and Jean-Jacques Vandewalle. The web of things: interconnecting devices with high usability and performance. In ICESS 2009, 2009.

[38] The World Wide Web Consortium. Oracle HVAC device model TD. https://github.com/w3c/wot/blob/master/plugfest/2018-bundang/TDs/Oracle/HVAC2.jsonld, 2018. Online; accessed 7 August 2018.

[39] The World Wide Web Consortium. BMW-X5 TD. https://github.com/w3c/wot/blob/master/plugfest/2018-bundang/TDs/EURECOM/bmw-x5.jsonld, 2018. Online; accessed 7 August 2018.

[40] W3C. BMW-S7 TD. https://github.com/w3c/wot/blob/master/plugfest/2018-prague/TDs/EURECOM-TD/EURECOM_BMW_S7_TD.jsonld, 2018. Online; accessed 8 August 2018.

[41] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 44(2):35–40, 2010.

[42] Dae-Man Han and Jae-Hyun Lim. Smart home energy management system using ieee 802.15. 4 and zigbee. IEEE Transactions on Consumer Electronics, 56(3):1403–1410, 2010.

[43] Alaa A Qaffas, Alexandra I Cristea, and Mohamed A Mead. Lightweight adaptive e-advertising model. Journal of universal computer science (JUCS)., 24(7):935–974, 2018.

[44] Simon Mayer, Ruben Verborgh, Matthias Kovatsch, and Friedemann Mattern. Smart configuration of smart environments. IEEE Transactions on Automation Science and Engineering, 13(3):1247–1255, 2016.

[45] M Balaji, E Aarthi, K Kalpana, B Nivetha, and D Suganya. Adaptable and reliable industrial security system using pic controller. May-2017, International Journal for Innovative Research in Science & Technology, (3):56–60, 2017.

[46] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. IEEE Transactions on industrial informatics, 10(4):2233–2243, 2014.

[47] You Jing, Zhang Lan, Wang Hongyuan, Sun Yuqiang, and Cao Guizhen. Jmeter-based aging simulation of computing system. In Computer, Mechatronics, Control and Electronic Engineering (CMCE), 2010 International Conference on, volume 5, pages 282–285. IEEE, 2010.

[48] Songze Li, Mohammad Ali Maddah-Ali, Qian Yu, and A Salman Avestimehr. A fundamental tradeoff between computation and communication in distributed computing. IEEE Transactions on Information Theory, 64(1):109–128, 2018.

[49] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. Communications of the ACM, 59(11):56–65, 2016.

[50] Roy T Fielding, Richard N Taylor, Justin R Erenkrantz, Michael M Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. Reflections on the rest architectural style and principled design of the modern web architecture (impact paper award). In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 4–14. ACM, 2017.

[51] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. Scientific american,

284(5):34–43, 2001.

[52] Ramanathan V Guha, Dan Brickley, and Steve Macbeth. Schema. org: evolution of structured data on the web. Communications of the ACM, 59(2):44–51, 2016.

[53] JSON Schema. JSON Schema | The home of JSON Schema. `http://json-schema.org/`. Online; accessed 24 August 2018.

[54] Tim Bray. The javascript object notation (json) data interchange format. Technical report, 2017.

[55] Tim Berners-Lee. Linked data, 2006, 2006.

[56] Ege Korkan. thingweb-playground. `http://plugfest.thingweb.io/playground/`. Online; accessed 24 August 2018.

[57] Eclipse Foundation, mkovatsc. thingweb.node-wot. `https://github.com/eclipse/thingweb.node-wot`, 2018. Online; accessed 9 August 2018.

[58] Node.js Foundation. Node.js. `https://nodejs.org/en/`. Online; accessed 9 August 2018.

[59] Raspberry Pi. model b. Raspberrypi. org. Saatavissa: https://www. raspberrypi. org/products/raspberry-pi-3-model-b/. Hakupäivä, 6:2018, 3.

[60] The World Wide Web Consortium. Web of Things (WoT) Thing Description September 2017. `https://www.w3.org/TR/2017/WD-wot-thing-description-20170914/`, 2017. Online; accessed 30 August 2018.

[61] Jan Lauinger. Testbench. `https://github.com/jplaui/testbench`. Online; accessed 31 August 2018.

[62] Jan Lauinger and Ege Korkan. Thing Description based Test Bench. `https://www.youtube.com/watch?v=BDMbXZ2O7KI`, 2018. Online; accessed 25 September 2018.

[63] Ruben Verborgh, Vincent Haerinck, Thomas Steiner, Davy Van Deursen, Sofie Van Hoecke, Jos De Roo, Rik Van de Walle, and Joaquim Gabarro. Functional composition of sensor web apis. In SSN, pages 65–80, 2012.

[64] Richard E Bellman. Adaptive control processes: a guided tour, volume 2045. Princeton university press, 2015.

[65] Victor Charpenay, Sebastian Käbisch, and Harald Kosch. Introducing thing descriptions and interactions: An ontology for the web of things. In SR+ SWIT@ ISWC, pages 55–66, 2016.