UNIVERSITÄT
MANNHEIM

An Interoperability Framework for Pervasive Computing Systems

Inauguraldissertation

zur Erlangung des akademischen Grades
eines Doktors der Wirtschaftswissenschaften
der Universität Mannheim

vorgelegt von

Felix Maximilian Roth

aus Stuttgart

Dekan:	Prof. Dr. Dieter Truxius
Erstreferent:	Prof. Dr. Christian Becker
Zweitreferent:	Prof. Dr. Philippe Lalanda
Tag der Disputation:	29. November 2018
Prüfungsausschuss:	Prof. Dr. Christian Becker (Vorsitzender)
	Prof. Dr. Hartmut Höhle

Abstract

Communication and interaction between smart devices is the foundation for pervasive computing and the Internet of Things, where users are surrounded by numerous computational devices communicating with each other and supporting users in their daily tasks. Smart devices provide value-added services to applications, and consequently to users. Pervasive platforms, that support developers in building new services and applications, have been extensively researched in the past. Nowadays, a multitude of different pervasive platforms exist. However, among further dissimilarities, they employ diverse protocols and interaction models, which prevents inter-platform communication. In real-world deployments, where usually more than one platform is present, this leads to the formation of platform-specific silos. Therefore, the need for interoperability between such platforms arises. Under those circumstances, several frameworks have been proposed targeting developer support for alignment of protocols and/or messages between pervasive platforms.

Although several interoperability frameworks exist, they do not address all issues that prevent inter-platform communication and, additionally, are often tailored to specific cases. For this reason, this thesis presents a framework which addresses all of those issues and allows for extension and customisation of different aspects, including new platforms as well as transformation mechanisms. The framework bases on uniform abstractions that support seamless translations of different features, such as service discovery, service access, and notification management, among others. The transformation model provides an automatic transformation mechanism, that can be easily extended or changed, as well as a manual transformation mechanism, that requires code writing. For evaluation, a prototype is implemented and assessed, providing support for six very distinct pervasive platforms. In particular, the feasibility of the proposed framework is demonstrated with three realistic scenario implementations, an effort evaluation, as well as a cost evaluation.

Acknowledgements

In the first place, I am deeply grateful to my advisor Prof. Dr. Christian Becker for his constant support, critique, and encouragement that made the creation of this thesis possible. Christian, thank you for giving me the opportunity to work in your research group, for always having an open door, and for invaluable discussions and conversations – work-related as well as personal. Furthermore, I want to thank you for sharing your knowledge in food and wine, whether at the chair, at Dobler's, in Shanghai, on Hawaii, or elsewhere.

I would further like to thank Prof. Dr. Philippe Lalanda for his input and willingness to act as second reviewer. Philippe, thank you for priceless challenging and thought-provoking discussions that greatly helped advancing my work. Also, I am heavily grateful to you for inviting me to Grenoble for two exceptional research stays permitting close collaboration with you and your team.

I would like to thank Prof. Dr. Hartmut Höhle for finding the time to join the board of examiners, as well as for wine recommendations and deliveries from the Bourgogne.

A special thanks is due to Prof. Dr. Gregor Schiele for his interesting and vivid lectures during my Master studies that, in the end, led up to pursuing my PhD at Christian's chair.

Further, I would like to thank all the people I had the pleasure of working with throughout the years at the chair, namely Dr. Patricia Arias-Cabarcos, Martin Breitbach, Janick Edinger, Kerstin Goldner, Benedikt Kirpes, Sonja Klingert, Dr. Christian Krupitzer, Markus Latz, Jens Naber, Martin Pfannemüller, Dr. Vaskar Raychoudhury, Dominik Schäfer, Dr. Sebastian VanSyckel, and Anton Wachner. I certainly enjoyed working with friends, and not just colleagues. Especially, I want to thank the 'little Christian' for supporting and mentoring me with basically everything when I first started at the chair. Thanks to Dom for giving an ear to me without getting the needle at the final stage of writing my thesis, as

well as for entertaining jam sessions. Thanks to Janick for always brightening the day with – sometimes funny – puns. Thanks to Jens for being a fellow sufferer with respect to BASE. Additionally, I owe special thanks to Patricia for her valuable comments and support at finalising my thesis.

Also, I would like to thank all the people I had the pleasure of working with at Grenoble, namely Colin Aygalinc, Pierangelo Castillo-Mora, Eva Gerber-Gaillard, and German Vega. Especially, I want to thank German for fruitful discussions and coding sessions. Moreover, thanks to German and Pierangelo for helping with the use case implementations.

This work was supported by the German Research Foundation (DFG) under grant BE 2498/9-1 'Interoperable Pervasive Systems'. I would like to thank each thesis student involved in the project for their contributions, namely Todor Angelov, Jia Liu, Alica Momann, Johannes Müller, Charlotte Stein, and Felix Strasser.

Last but not least, I would like to thank my family and friends for always being there for me. To my parents, Marina and Michael, thank you for your unconditional support and caring. To Dr. Moritz, thank you for being such an outstanding brother, quite likely the best. To my grandparents, Irma and Albert, thank you for still keeping your fingers crossed for me at an age of over 90 years. To my girlfriend, Lara, thank you for being there for me and motivating me, regardless of how many hundreds of kilometres in between us.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CS	Client-server
DHCP	Dynamic Host Configuration Protocol
ESB	Enterprise Service Bus
GB	Gigabyte
GHz	Gigahertz
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
ID	Identifier
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
LLOC	Logical Lines of Code
MQTT	Message Queue Telemetry Transport
PS	Publish-subscribe
REST	Representational State Transfer
SAWSDL	Semantic Annotations for WSDL
SOAP	(originally) Simple Object Access Protocol
SOA	Service-oriented Architecture
SOC	Service-oriented Computing
SQL	Structured Query Language
TCP	Transmission Control Protocol
TS	Tuple Space
UPnP	Universal Plug and Play
WS-BPEL	Web Services Business Process Execution Language
WSDL	Web Service Description Language
XML	Extensible Markup Language

1. Introduction

This chapter introduces the present thesis with a motivation, the research questions and contributions, and the structure of this work. Subsequently, Chapter 2 presents the theoretical background that is required for the remainder of the thesis. Parts of this chapter are based on [147][1] and [149][2].

1.1. Motivation

Pervasive computing [170] promotes the integration of smart, networked devices into everyday environments in order to provide added-value services to people. The development and administration of such pervasive services is especially complex for several reasons. Services are based on dynamic, heterogeneous resources (e.g., devices and networks) over which they do not have control. Thus, these services are in charge of adapting to their environment, and not the other way around. Also, services are intended to be unobtrusive, requiring minimal attention and intervention from the service recipients (or users). The most administration, therefore, has to be performed autonomically by the services themselves. Moreover, stringent non-functional requirements related to security or privacy, for instance, must be achieved.

In order to ease the development and administration, pervasive middleware platforms have been developed. They provide a development model and a set of technical services. These technical services include, among others, means for communication between services and applications. This way, developers are not distracted by sideshow issues, but can focus on application development. Today, pervasive platforms are well accepted and used in several industrial (e.g., Bosch[3],

[1] [147] is joint work with M. Pfannemüller, C. Becker, and P. Lalanda
[2] [149] is joint work with G. Vega, C. Becker, and P. Lalanda
[3] https://www.bosch-smarthome.com/uk/en/home

Samsung[4], or Panasonic[5]) and academic settings (e.g., BASE [12] or iPOJO [50]).

Although these platforms are well accepted, their multitude nowadays hurts the development of rich services in large, distributed environments, like multi-apartment buildings or towns, due to the formation of platform-specific silos. Indeed, such environments are characterised by the presence of different platforms and devices using distinct technologies and architectures. Many smart devices, and consequently platforms, are already installed, e.g., in smart homes, smart offices, or smart factories. Thus, it is impractical and impossible to replace those platforms by one common system [65]. If one platform prevailed among the existing ones, interoperability between these platforms would not be necessitated [144]. However, this is not expected due to the fact that commercial providers want to keep users in their silos [157]. Through interworking, different platforms can synergise and offer more flexibility to the user since a greater variety of devices is available [118]. For this reason, several projects have been launched to improve interoperability between platforms in view of building large, heterogeneous pervasive environments (see, for instance, the European initiative called IoT-EPI[6] promoting several projects on interoperability, e.g., BIG-IoT [27]).

Achieving interoperability between pervasive platforms is a complex task due to various discrepancies between them. In particular, these discrepancies include the use of distinct technologies [141], service discovery mechanisms [57], and interaction models [16], as well as differences in their data representations [21], service interfaces [86], non-functional properties [21], and update notifications [147]. Building a solution that solves all of these disparities can rapidly become hardly comprehensible and understandable. Therefore, developers need to be supported through a framework. However, existing frameworks do not address all of the mentioned issues and, further, only provide *ad hoc* solutions. In this thesis, an interoperability framework is proposed that is able to manage all of the identified heterogeneous characteristics of pervasive platforms. Additionally, it provides development support in order to easily extend and customise solutions, including the integrated alignment algorithms. This way, tailored interoperability solutions can be realised, and future platforms and services can be incorporated.

[4]https://www.samsung.com/us/smart-home/smartthings/
[5]https://www.panasonic.com/uk/consumer/smart-home.html
[6]http://iot-epi.eu

1.2. Research Questions

Following the motivation, the main objective of this thesis is the development of an interoperability framework that, on the one hand, is able to address the whole set of identified heterogeneities and, on the other hand, provides support for customisation of the framework's solution instances. For this, an interoperability framework must provide means to align protocols and messages, while at the same time it has to be extensible and flexible with respect to the integration of new platforms, services, and even algorithms. Therefore, this thesis will answer the following research questions:

1. What is the state of research in interoperability frameworks for pervasive computing systems?
2. How can different pervasive platforms be abstracted in a uniform fashion in order to support interoperability?
3. How can an interoperability framework support developers in building custom interoperability solutions?
4. What are the costs for achieving interoperability?

1.3. Contributions

This thesis presents a general framework for interoperability between pervasive computing systems, entirely developed and evaluated on realistic use cases. Its main contributions are as follows:

First, an evaluation framework is developed for a thorough analysis of existing interoperability frameworks. It includes a set of heterogeneities that prevent pervasive platforms from co-operating, a set of interoperability solution designs, and a set of interoperability framework requirements. Existing approaches then are assessed with the evaluation framework.

Second, uniform abstractions are developed for messages, services, service discovery, service access, and notifications. These abstractions are based on commonalities between different interaction paradigms and support the alignment between different platforms.

Third, an integrated alignment process is presented for overcoming the identified heterogeneities. Unlike in other approaches, here, the alignment process allows for its easy extension and customisation. Therefore, it provides an automatic approach for transformation of simple service types, as well as a manual approach where developers are supported in writing code for transformation purposes. Developers can choose to use the automatic or manual tool per platform and service.

Fourth, a prototype is implemented that integrates a reference architecture with re-usable components. The prototype integrates the previous contributions and allows for validation. Further, six diverse pervasive platforms are integrated in the prototype by using or adjusting the re-usable components, or writing new ones.

Finally, the prototype implementation is extensively evaluated. Therefore, a proof of concept is provided based on three realistic smart home scenarios. Furthermore, a requirements evaluation, an effort evaluation regarding the overhead for developers, as well as a cost evaluation take place. The evaluation underlines the feasibility of the proposed framework and its reasonable costs for providing interoperability.

1.4. Structure

The remainder of this thesis is structured as follows. Chapter 2 gives an overview on the theoretical background of the thesis. Next, Chapter 3 develops an evaluation framework which subsequently is used to assess existing interoperability approaches. Afterwards, Chapter 4 details the system model of this thesis prior to presenting an interoperability framework for pervasive computing systems. Chapter 5 gives details on the prototype implementation which then is evaluated in Chapter 6. Finally, Chapter 7 closes the thesis with a conclusion and an outlook on future work.

2. Background

This chapter provides background information on fundamental terminologies required for this thesis. First, different computing paradigms are presented in Section 2.1. Second, Section 2.2 introduces various interaction models since they are one critical aspect impeding interoperability. Last, for the same reason, Section 2.3 familiarises the reader with notification systems. Parts of this chapter base on [147] and [149].

2.1. Computing Concepts

Most pervasive middleware platforms focus on the notion of service-oriented computing [130] in order to improve interoperability. Service-oriented computing facilitates a 'dynamically adaptable architecture by supporting runtime evolution and modification of each service independently' [97]. This architectural adaptability is crucial in pervasive and Internet of Things applications to drive seamless adaptations when the environment or requirements change. Therefore, this section presents fundamentals in service-oriented computing, pervasive computing, the Internet of Things, and further related concepts.

2.1.1. Service-oriented Computing

Service-oriented computing (SOC) [130] is a widely used concept [96] in pervasive computing, the Internet of Things, and further related paradigms, e.g., ambient computing. In SOC, services are the fundamental elements of an application [130]. Services are re-usable, modular units that implement business logic for certain functionalities, e.g., a light service may offer a functionality to change the state of a light (on/off). Applications integrate and/or are composed of one or several services, i.e., they make use of services in order to perform their task. For example, a simple light application, requiring a light service and a presence

service, turns the light on if a person is present, and turns the light off if no person is detected any more. Applications bind to the required services at runtime, and not at design time like in classical software development. This is called *late binding* [130]. Therefore, services offer well-defined interfaces such that provided and required functionality can be specified adequately. Furthermore, the internal business logic of a service is not known and also not important for an application. This enables a separation between service functionality (interface) and service implementation (business logic) [96]. New applications can be easily built or old ones can be adapted by replacing service bindings at runtime [131], if they are stateless and have equivalent non-functional properties. Thus, the service-oriented computing paradigm provides a dynamic, flexible approach that eases software development [81].

Because service binding happens at runtime, a mechanism must ensure that services can be discovered and accessed by applications or other services. Service discovery and access are provided through a service-oriented architecture (SOA) [130]. A SOA describes relationships and interactions between its three actors: service consumer, service provider, and service registry. The *service consumer* [27, 140] (also service client, e.g., [96, 130], or service requestor, e.g., [138, 173]) seeks to find and use one or more services. The *service provider* [96, 130] offers one or more services. The *service registry* [96, 130] (sometimes also service directory, e.g., [171, 173], or service broker, e.g., [50, 96]) stores service descriptions of registered services and interacts with service consumers as well as service providers. The basic interactions between these actors are as follows. A service provider publishes its service description to the service registry. The *service description* includes information on the provided functionality, possibly offered non-functional properties, as well as the service grounding [110], i.e., how a service can be accessed with respect to its location and protocols. A service consumer asks the service registry for a certain service and, if such a service is present in the system, the service registry returns the service description. The service request contains the required functionality and, possibly, required non-functional properties, e.g., security requirements. Then, the service consumer can directly access the service provider, i.e., it can use the interfaces and service grounding provided in the service description to interact with the service provider. Figure 2.1 visualises these relations and interactions.

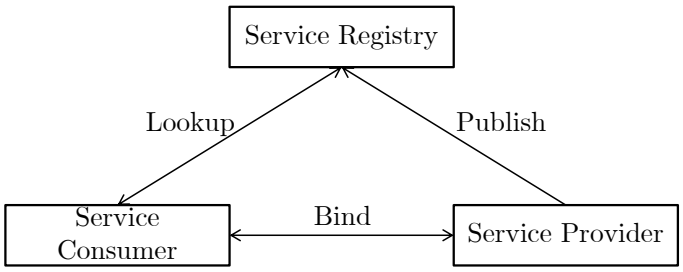


Figure 2.1.: Logical View on a Service-oriented Architecture (cf. [130]). Service providers can publish their services to the service registry. Service consumers can look services up at the service registry. Service interaction happens directly between consumers and providers.

Services can be categorised into simple services and composite services [130]. Simple services offer stand-alone functionality, i.e., they encapsulate all required business logic themselves (e.g., the mentioned light service). Composite services make use of other services in that their offered functionality can only be realised by accessing further services. For example, an economic heating service might make use of a window state service to check if the window is closed before it regulates to a higher temperature. Composite services can be the result of orchestration or choreography. The interested reader is referred to [48] for more information on orchestration and choreography. Furthermore, an application and a composite service are similar in that both make use of other (simple or composite) services. However, an application does not publish or offer a well-defined interface (or service description) for accessing its functionality. Thus, service consumers can be either applications or composite services. Besides, an *entity* is a device that acts as provider, consumer, or both.

Moreover, the process of discovering services is called *service discovery*. It includes all activities until a consumer is able to communicate directly with the service, i.e., advertisement of service descriptions by providers, lookup requests and responses between consumer and service registry, as well as service matching at the service registry. Then, a consumer possibly wants to use (or access) a discovered service. This process is called *service access*. According to [130], services need to meet certain requirements to ensure the working of these mechanisms: technology neutrality, loose coupling, and location transparency. *Technology neutral* means that a commonly accepted standard should be used to ease

service discovery and invocation. *Loose coupling* is that the service implementation is transparent to the consumer as the internal processing is not important to him/her. *Location transparency* says that providers should be accessible regardless of their location.

In order to simplify service and application development in SOC, different protocols have been developed [144]. Service discovery protocols enable automatic publication and discovery of services without previous knowledge and with minimal human effort [57, 96]. Also, service access protocols enable transparent service access. Service-oriented *middleware platforms* that support developers in creating services and applications usually include service discovery as well as service access protocols. Many different service discovery protocols, e.g., service location protocol (SLP) [75] and simple service discovery protocol (SSDP) [32], service access protocols, e.g., SOAP [26], and middleware platforms, e.g., Jini [5], BASE [12], or iPOJO [50], exist nowadays. Naturally, by relying on the SOA, those solutions satisfy the three requirements introduced in [130]. However, each of those solutions realises the SOA in a different way, preventing services and applications developed upon different middleware platforms from co-operating. A more detailed overview on reasons that hinder services that are developed with different middleware platforms to interact with each other can be found in Section 3.1. For reasons of simplicity, hereafter, sometimes 'an entity that is developed with pervasive platform A' is abbreviated by 'an entity of platform A'.

Other distributed computing paradigms, such as pervasive computing or the Internet of Things, often make use of the service-oriented computing paradigm [6]. Those are explained in the following sections.

2.1.2. Pervasive Computing

Pervasive computing, or also ubiquitous computing, was first introduced by Mark Weiser in 1991 [170] who envisioned a change from traditional desktop computing to the modern computing landscape where the physical environment is equipped with computational devices that are communicating with each other and are interwoven with artefacts of the everyday life. These devices further provide functionality to users in order to seamlessly support them in their everyday tasks, and thus get more ubiquitous every day [170]. Satyanarayanan also describes this

as 'technology that disappears' [151]. It becomes clear that pervasive computing is user-centric. In order to provide meaningful functionality to the user, it makes use of the context. Context is often defined as 'any information that can be used to characterize the situation of an entity' [43] whereby an entity is a 'person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves' [43]. In other words, an entity can be any kind of object that may have an influence on the application's behaviour. Context can be, for instance, the temperature in a room, persons nearby, or also the battery status of a device. In a simple scenario of an automatic door, the context may only include the presence of a person in front of the door. In pervasive computing, applications do not only consider the context to provide user-related functionality, but they are also able to alter the context. For instance, when the user starts playing a film on the television in the evening, the lights may be automatically dimmed to maximise the film experience.

Pervasive computing builds upon distributed computing and mobile computing [151]. Distributed computing permits computers to be connected over a network and share functionalities, whereas mobile computing targets at mobile devices having a network connection 'anytime anywhere' [150]. Pervasive computing extends the goal of mobile computing to a proactive information access 'all the time everywhere' [150].

Building upon mobile computing [150, 151], pervasive environments entail dynamism due to user and device mobility. This means that pervasive computing environments are volatile in terms of devices and services sporadically entering and leaving the environment. This characteristic predestines the use of the SOC paradigm for pervasive computing as it shifts service binding from design time to runtime. Raverdy *et al.* even claim that only the SOC paradigm has made the realisation of the pervasive computing vision possible [142].

The technical realisation of a pervasive computing scenario is attained by a pervasive system [107]. A *pervasive system* consists of a set of devices and a set of users which are located in a physical space. Users interact with the devices. Devices can be traditional devices, e.g., personal computers and mobile devices, or smart devices, e.g., sensors or actuators [150]. Mobile devices include smartphones, laptops, and alike. Sensors help to measure context, whereas actuators can be used to realise context adaptations. They are connected through network-

ing means and provide functionality, in form of services, to users through pervasive applications. A *pervasive application* is an application that uses and may alter resources and context of the current physical space. Thus, it is context-aware (through sensors) and context-altering (through actuators) [107]. Pervasive applications are usually implemented with the help of a pervasive middleware platform (henceforth called pervasive platform). A *pervasive platform* assists in application development and administration [49]. Therefore, it offers a development model and technical services to developers and administrators. These technical services may include communication, context management, and conflict management. Many different pervasive platforms exist for realising pervasive computing scenarios and, nowadays, most of them are based on the SOC paradigm, e.g., BASE [12], iPOJO [50], ubiSOAP [33], SAI [128], nSOM [53], DigiHome [146], or AutoHome [25], in order to attain flexibility and dynamism. Figure 2.2 summarises the notion of a pervasive system. Besides, pervasive systems have usually been built as closed systems for one specific purpose.

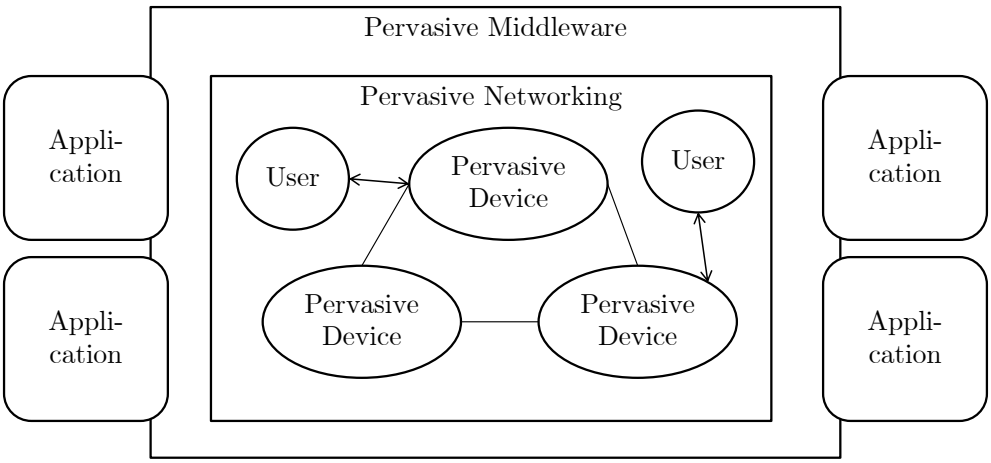


Figure 2.2.: Pervasive System (cf. [150]). Devices offer functionality to users and communicate with other devices. Applications are implemented with the help of a pervasive middleware.

In industry and academia, many different application areas for pervasive computing have emerged in different domains, such as home (e.g., [78]), healthcare (e.g., [166]), assisted living (e.g., [134]), office (e.g., [2]), entertainment (e.g., [22]), manufacturing (e.g., [174]), logistics (e.g., [109]), or agriculture (e.g., [168]).

Pervasive computing further is closely related to the Internet of Things.

2.1.3. Internet of Things

The *Internet of Things* (IoT) can be defined as 'the worldwide network of interconnected objects uniquely addressable based on standard communication protocols' [10]. A *thing* (or also object) is actively participating and interacting within application domains, i.e., it may distribute sensed information or react to upcoming events [159]. Thus, the IoT vision extends pervasive computing on a global scale [91, 114]. According to [73], there will be 24 billion potentially interconnected devices in the world, which is triple the world's population nowadays. This fact is compatible with the IoT (and pervasive computing) vision where users are surrounded by a multitude of devices located anywhere around them.

For developing IoT applications, developers make use of a middleware platform. Equivalently as in pervasive computing, SOC is the preferred paradigm of these middleware platforms [6, 73]. Things can be seen as devices providing services [87, 161] (in this domain also called resources, e.g., [87, 160]). Another commonality between IoT and pervasive computing is that early IoT systems were built as closed systems, preventing them from inter-system communication [91]. Today, this is still the case because IoT product vendors use their own developed platforms [65] trying to force users into their ecosystem after their first IoT product purchase.

Furthermore, since IoT is seen as an extension to pervasive computing, their application domains also overlap. They include [7, 73, 159, 172]: home, enterprise, community, city, country, agriculture, water, transportation, logistics, healthcare, energy, aviation, automotive, telecommunications, pharmaceutics, manufacturing, entertainment, insurance, and recycling.

In the following, further related concepts to pervasive computing and the Internet of Things are briefly discussed.

2.1.4. Related Concepts

Pervasive computing (and thus transitively also IoT) is closely related with context-aware computing, wearable computing, and ambient computing. *Context-aware computing* [152] deals with applications that can adapt their behaviour to the current context. Therefore, sensors can measure relevant context attributes

and make them available to applications. An example is that a smartphone display changes its brightness depending on the environmental lighting conditions. Pervasive applications are also context-aware and further context-altering. This means that they do not only adapt themselves to the context, e.g., by increasing the display brightness, but they also adapt the context according to their own requirements, e.g., by switching the lights off instead of increasing the display brightness. *Wearable computing* is concerned with the miniaturisation of networked computing devices in order to wear them on the body [108]. Examples are smartwatches, head-mounted displays, or fitness trackers. Consequently, wearable computing directly contributes to achieve the pervasive computing vision. Besides, *ambient computing* also is about the seamless integration of computing devices in the real world [116]. However, it focuses on an intuitive and natural user interaction.

The IoT relates to the Web of Things. Like IoT, the Web of Things uses Web standards (e.g., REST, HTTP, JSON) in order to make networked everyday objects, e.g., thermometers or heaters, available through the Web [74]. Yet, it focuses on the application layer, whereas IoT is concerned with establishing connectivity between devices through the Internet.

As mentioned before, there exists a variety of middleware platforms for pervasive computing and IoT – and their related concepts – that support developers in implementing new applications and services. Though, interaction between services and applications is usually only possible in their own domain, i.e., inter-platform interaction is not feasible. The use of distinct interaction models by those platforms is one reason for this [16]. Therefore, the next section gives an overview on the most common interaction paradigms.

2.2. Interaction Models

The three most prominent interaction models are [68]: client-server, publish-subscribe, and tuple space. The following paragraphs explain those interaction paradigms in more detail. Also, the paradigms are briefly discussed with respect to service discovery and, furthermore, space coupling, time coupling, and synchronisation coupling. *Space coupling* is about the identification of sender and

receiver [51]. It is loose if sender and receiver do not know each other, i.e., they do not hold references of each other in order to communicate. This can be achieved with a mediating entity. Otherwise, space coupling is tight. *Time coupling* describes the time of presence and availability of sender and receiver [51]. It is loose if interacting entities do not need to be connected at the same time for interaction. Again, a mediating entity can help to achieve this. Else, time coupling is tight. *Synchronisation coupling* determines if sender and receiver are blocked during communication [51]. It is loose if sender and receiver are not blocked during communication, i.e., the sender is not blocked until it receives a reply and the receiver can communicate with several entities at a time. Otherwise, synchronisation coupling is tight; this is also called *synchronous communication*.

2.2.1. Client-server Interaction

In the client-server (CS) paradigm, a server provides one or more services, whereas a client might use these services. In that case, the client sends a request to the well-known server which processes the request in response. Then, if the client expects a result, the server sends it to the client.

The Web Service Description Language (WSDL) specification [40] defines four operations for CS interaction: one-way message, notification, request-response, and solicit-response. A *one-way message* is a message from client to server, while a *notification* is a message from server to client. A *request-response* operation is a request from the client to the server which processes the request and delivers a response. A *solicit-response* operation resembles the request-response operation, but the roles of client and server are interchanged. Basically, one can classify these operations as one-way interaction (one-way message and notification) and two-way interaction (request-response and solicit-response) [23]. Henceforth, they are subsumed as *one-way message* and *request-response* operations, respectively. The CS interaction scheme is summarised in Figure 2.3. There, and also in the remainder, the notions of client and server are replaced by (service) consumer and (service) provider, conforming to the terminology introduced in Section 2.1.1.

CS interaction is tightly coupled regarding space because provider and consumer have to know each other for the purpose of communication. Furthermore, both provider and consumer must be present and available at the same time, in-

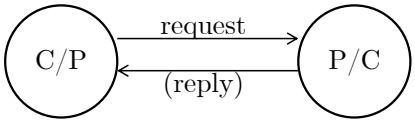


Figure 2.3.: Client-server Interaction. One entity sends a request which is processed by the other entity and a result is possibly returned (C - consumer, P - provider).

dicating a tight time coupling. The consumer is usually blocked until it receives the reply. Hence, synchronisation coupling is tight. Extensions to the classical CS model are possible, e.g., using a message queue for loose time and space coupling [21] or a callback for loose synchronisation coupling [90]. Moreover, platforms that base on the CS paradigm usually employ a SOA for service discovery, as explained in Section 2.1.1 (see, e.g., BASE [12] or iPOJO [50]).

2.2.2. Publish-subscribe Interaction

A publish-subscribe (PS) interaction [51] requires three actors: subscriber, publisher, and broker. Subscribers can subscribe to certain events at the broker. Publishers can publish events to the broker. The broker checks whether published events match subscriptions and, if so, forwards them to the relevant subscribers.

The two basic operations are: subscribe and publish. A *subscribe* operation contains information on the kind of events in which the subscriber is interested in form of a filter. A *publish* operation includes the event itself and possibly some event category. The event matching mechanism depends on the scheme of filter and event category. This is discussed in detail in Section 2.3. So far, one can think of the filter and event category as brief event description, e.g., *Temperature* or *Humidity*. In case that the filter equals the category, the event is forwarded to the respective subscriber. Figure 2.4 summarises the PS interaction scheme. There, and also hereafter, the terms subscriber and publisher are replaced by (service) consumer and (service) provider, respectively.

Space coupling is loose for the PS interaction model since providers and consumers only interact with the broker, and not with each other. Hence, providers do not know the recipient of its events and consumers do not know the provider of received events. Also, time coupling is usually loose because neither provider

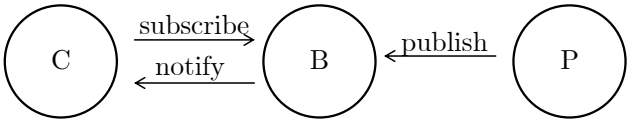


Figure 2.4.: Publish-subscribe Interaction. Consumers (C) can subscribe for events, providers (P) can publish events, the broker (B) checks for matches and forwards events, if appropriate.

nor consumer have to be available at the same time. The consumer can be absent at the time an event is published. When it subscribes for this event at the broker, the broker delivers the event. Furthermore, synchronisation coupling is loose since neither provider nor consumer are blocked during transmission. According to [137] and [51], in many PS-based platforms perform, providers advertise categories of events they potentially publish. Actually, these advertisements are only meant for the broker for event routing optimisation [137], but can be also used for service discovery (see Section 4.3.2).

2.2.3. Tuple Space Interaction

In tuple space (TS) interaction [67], entities communicate via a shared TS by adding and withdrawing tuples. A tuple is a finite sequence of data elements, e.g., the tuple ('lighstate', 'on') indicates that the light is on. A TS is a shared memory in which data is represented as tuples. For communication, an entity writes a tuple to the TS, where it stays as long as no entity withdraws the tuple. Entities withdraw or read tuples by using tuple templates, e.g., the template ('lightstate', ?) can be used to receive the above tuple. A tuple template describes a tuple with actuals – concrete values, e.g., 'lightstate' – and formals – wild cards, represented by '?'. Then, the TS looks for tuples that match the template. At this, actuals have to match exactly the tuple value, whereas formals can be any value.

The three basic operations of TS interaction are [67]: out, in, take. The *out* operation, for adding a tuple to the TS, requires a tuple and has no return value. The *take* operation, for withdrawing a tuple, requires a tuple template and returns a tuple. The *in* operation is like the take operation, but it reads a tuple without removing it. Figure 2.5 shows the TS interaction scheme. There, and also in the remainder, a provider is an entity that adds tuples to the TS, whereas a consumer is an entity that reads/takes tuples from the TS.

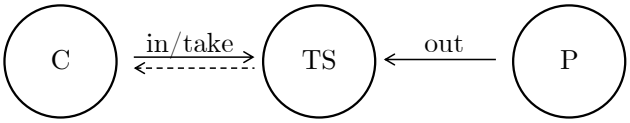


Figure 2.5.: Tuple Space Interaction. Providers (P) add tuples to the TS, while consumers (C) access them by using tuple templates.

TS interaction is loosely coupled regarding space. Providers add tuples to the TS, whereas consumers take/read tuples from the TS. Thus, provider and consumer do not know each other. As well, time coupling is loose because neither provider nor consumer have to be available at the same time. Synchronisation coupling is loose on the provider side. However, the consumer is blocked until it receives a matching tuple for a take/in operation. Regarding service discovery, devices often send advertisement tuples containing information on their provided services. Other entities can read out those tuples from the TS by using an appropriate template in order to know about present services. Nevertheless, not all platforms provide such a mechanism, e.g., Lime [136] does, whereas Limone [60] does not.

2.2.4. Overview

Table 2.1 provides an overview on the differences between these interaction models with respect to space, time, and synchronisation coupling, as well as service discovery. It is obvious that the disparate characteristics and service discovery mechanisms prevent platforms that use distinct interaction paradigms from interacting.

Model	**Space coupling**	**Time coupling**	**Synchronisation coupling**	**Service discovery**
CS	tight	tight	tight	SOA
PS	loose	loose	loose	event category advertisements
TS	loose	loose	loose on provider side	advertisement tuples

Table 2.1.: Interaction Model Characteristics. Differences in space, time, and synchronisation coupling, also lead to discrepancies in service discovery mechanisms.

Furthermore, dissimilarities in their notification mechanisms hinder the interworking of distinct platforms [149]. For this reason, the next section gives an overview on notification systems.

2.3. Notification Systems

Pervasive platforms sometimes allow for notifications, independent of their interaction model. In the context of pervasive computing and IoT, a *notification* (or also *event*) is a piece of information in that consumers might be interested. Delivery of notifications is often performed using the PS interaction paradigm. Basically, notification systems and the PS interaction model are highly correlated. However, whereas PS interaction does include means to interact for performing a task, e.g., change a specific value, the purpose of notification systems is only to communicate updates, e.g., a value has changed. Thus, in this thesis, PS interaction is about service interaction, whereas notifications are about service updates. Section 2.2 introduced the PS interaction but omitted details on the filter and event category. The literature mainly distinguishes three different schemes for this purpose [37, 155]: channel-based, subject-based (or also topic-based [51]), and content-based. Among these schemes a trade-off between expressiveness and overhead exists [51].

In a *channel-based* system, the event category is represented by channels. Thus, an event is published to one specific channel, e.g., *Temperature*, *Light*, or *Humidity*. Consumers can subscribe to one or more of these channels. Subsequently, they will receive all events targeting one of the subscribed channels. Some exemplary channels are shown in Figure 2.6. On the one hand, channels have a low expressiveness. But on the other hand, they only require low overhead due to the simple event matching mechanism.

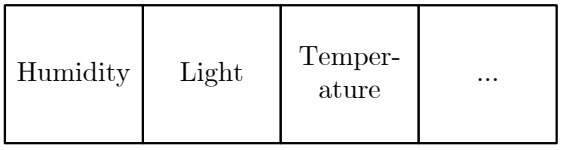


Figure 2.6.: Exemplary Channels. Consumers can subscribe to a specific channel in order to receive every event targeting this channel.

In a *subject-based* system, the event category is represented by subjects. A subject is part of a hierarchical categorisation. Figure 2.7 depicts an exemplary subject hierarchy. In this example, if subscribers are interested in the temperature, they subscribe to the subject *Physical Environment/Conditions/Temperature*. Here, '/' denotes the next lower level in the hierarchy. Furthermore, subject-based systems support the use of wild cards in order to subscribe to a sub-tree (here denoted by '#') or a specific level of the tree (here denoted by '+'), instead of only one subject. For example, subscribing to *Physical Environment/Conditions/#* will return each upcoming event that is tagged by a subject of the sub-tree, e.g,, *.../Temperature* or *.../Light/Level*. Subjects have a higher expressiveness compared to channels, but they also require a higher overhead.

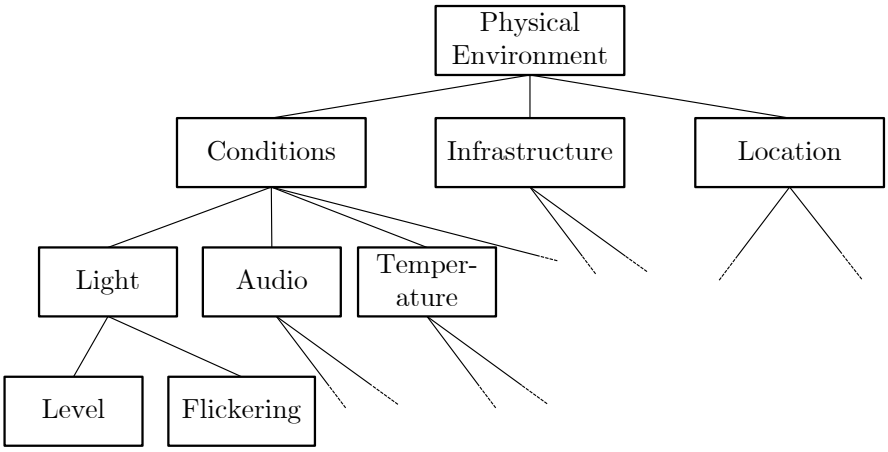


Figure 2.7.: Exemplary Subject-based Hierarchy (based on [154]). Several levels of subjects exist. Also wild cards are allowed in order to subscribe to a sub-tree or level of the hierarchy.

In a *content-based* system, event-subscription-matching bases on a data model with a corresponding filter model. These models are usually highly application-specific. Figure 2.8 shows an extract of a content-based data model. Consumers can subscribe to events that satisfy certain content requirements, such as *tempInfo.value > 20 and tempInfo.unit = 'Celsius'*. Thus, unlike the two former schemes, content-based systems do not use any additional event category. Moreover, these systems have a high expressiveness but also a high overhead.

Additionally, notification systems can be classified by their delivery modes [61]. On both sides – consumer and provider – event delivery can be realised

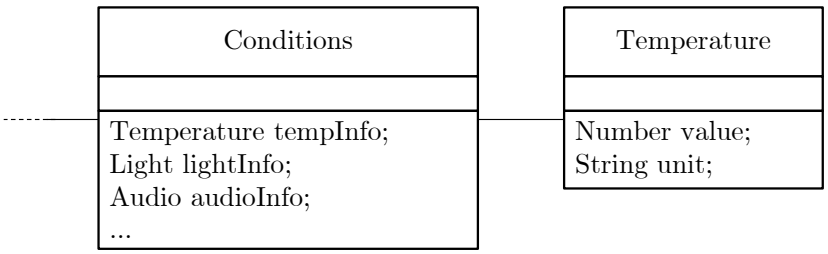


Figure 2.8.: Exemplary Content-based Data Model. Content-based systems require a complex, application-specific data model.

with a push or pull mechanism. On the consumer side, a push mechanism means that the notification system pushes events to the consumer, whereas a pull mechanism denotes that the consumer pulls for new events at the notification system periodically or sporadically. On the provider side, a push mechanism implies that the provider pushes events to the notification system, whereas a pull mechanism indicates that the notification system periodically or sporadically pulls for new information at the provider.

Altogether, this chapter discussed the theoretical background with respect to the service-oriented computing paradigm which often serves as foundation for pervasive and Internet of Things platforms. Furthermore, different interaction models and notification systems were reviewed. The next chapter builds upon these fundamentals while elaborating on interoperability issues, solution designs, and requirements for an interoperability framework, as well as related interoperability approaches.

3. State of the Art

The preceding chapter discussed the theoretical background. This chapter investigates the state of the art in interoperability frameworks. First, Section 3.1 develops an evaluation framework containing the addressing of heterogeneities between pervasive platforms, possible solution designs of interoperability frameworks, as well as general interoperability framework requirements. Subsequently, Section 3.2 performs a thorough literature analysis where existing interoperability approaches are assessed with the developed evaluation framework. Last, the approach in this thesis is demarcated from literature. Besides, this chapter is based on and extends the works in [147] and [149].

3.1. Evaluation Framework

Before having a look at the state of the art in interoperability frameworks for pervasive and IoT systems, one should be on familiar ground with the term interoperability itself. An often used definition of *interoperability* is given by the *IEEE Standard Glossary of Software Engineering Terminology* as 'The ability of two or more systems or components to exchange information and to use the information that has been exchanged' [83]. Although the given definition is quite old (it is from 1990), it clarifies the general understanding. In the context of pervasive computing, this means that service consumers should be able to discover and access any available service, i.e., communicate and understand the messages, independent of its platform. More recent definitions include issues that need to be overcome (e.g., 'Interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform' [169]), include how issues should be addressed (e.g., 'Ability of a system or a product to work with other systems or products without special effort on the part of the customer. Interoperability is made possible by the implementation of standards' [84]), or are very research field-specific (e.g., 'Interoperability characterises the

extent to which two software components from different manufacturers, which are functionally compatible, can be made to work together correctly by reconciling the differences in their interfaces and behaviours' [14]). However, as issues might be extended and new solutions may arise in future, these definitions do not seem well-suited. Therefore, in this thesis, the definition from [83] is used.

Having defined interoperability, the remainder of this section establishes an evaluation framework for assessing existing interoperability approaches.

3.1.1. Categorisation of Heterogeneities

Based on literature, the following issues have been identified that hinder interoperability in service-oriented pervasive environments: communication, (service) discovery, interaction paradigm, data, application, non-functional properties, and notifications. Hereafter, those issues are explained in more detail.

Communication heterogeneity (H1) [47, 96, 116, 141] arises if devices are using different *communication technologies* [47], such as Wi-Fi or Bluetooth. Since a Bluetooth device cannot communicate with a Wi-Fi device, communication between those devices is impossible without further means. Moreover, a different *management model* (infrastructure- or *ad hoc*-based) might prevent middleware platforms from communicating. An infrastructure-based model makes use of a neutral instance, e.g., a router, for exchanging messages, whereas in an *ad hoc*-based model devices directly exchange and forward messages between each other. Therefore, also header information of messages differs.

Discovery heterogeneity (H2) [3, 16, 29, 47, 57, 59, 71, 96, 118, 141, 164, 173] appears due to disparities in discovery mechanisms. This includes the use of different message syntaxes and semantics [57], i.e., formats and contents. Ganzha *et al.* additionally mention discrepancies in the used service models [65]. These points are consolidated as *service discovery language*. Moreover, disparate protocol behaviour can also lead to difficulties. For instance, UPnP [123] uses the simple service discovery protocol (SSDP) [32] which bases on the hypertext transfer protocol (HTTP), whereas BASE [12] employs a proprietary protocol using Java objects. Zhu *et al.* differentiate between further variations, such as initial communication method (unicast, multicast, or broadcast), discovery and registration

(query-based or announcement-based), discovery infrastructure (non-directory-based or directory-based), and service usage (explicitly released or lease-based) [173]. Here, *protocol behaviour* summarises these points.

Interaction heterogeneity (H3) [9, 16, 21, 71, 86] exists because platforms use distinct *interaction models* [86], e.g., client-server, publish-subscribe, or tuple space. These different models are not designed to work together (cf. Section 2.2). Furthermore, the specific *instantiations of these models* [71, 86, 96, 118] differ between pervasive platforms as diverse protocols and data formats are used.

Data heterogeneity (H4) [16, 21, 47, 86, 141] implies that the same information is expressed in different ways. There, a *syntactic mismatch* occurs due to different data formats [16, 21, 47, 86, 96, 141, 164], e.g., extensible markup language (XML) or a Java object. Hereby, also the carried information can be different. If the same information is expressed using a different vocabulary, such as `price` and `cost`, the heterogeneity is of *semantic nature* [16, 21, 86, 96, 118]. Further, data can be of different units among middleware platforms, e.g., the price can be indicated in Euro or Swiss Francs. This also counts into the semantic part of data heterogeneity. All in all, data heterogeneity is a very general heterogeneity that also appears in the subsequent heterogeneities, where it may not be explicitly mentioned.

Application heterogeneity (H5) [16, 86] emerges due to the fact that developers specify functionality differently with respect to the service interface. Thus, developers possibly implement *business operations* in various ways among platforms [85, 86], e.g., `getName` versus `getFullName`. Moreover, *operation granularity* can vary between platforms [85, 86], i.e., an operation on one platform might be implemented with two operations on another platform. For instance, it can happen that the operation `getName` on one platform is implemented with two separate operations `getFirstName` and `getLastName` on another platform.

Non-functional properties heterogeneity (H6) [21, 57] emerges from different syntactic (data types) and semantic (e.g., 'screen size' versus 'display size') data representation of non-functional properties. Furthermore, different *domains* of non-functional properties can exist among middleware platforms. For example, one platform supports the requirement 'security' and another platform supports the requirement 'reliability'. Those platforms cannot provide appropriate means

in order to fulfil the other one's requirements. Non-functional properties do also include context properties encountering the same problems.

Notification heterogeneity (H7) [149] arises from dissimilarities in the notification models of distinct platforms. The first issue is that some platforms do not even *provide/support notifications*. However, information of such platforms might be of interest to others. Hereafter, platforms that do support notifications are called *supporting platforms*, whereas platforms that do not are called *non-supporting platforms*. Two supporting platforms can differ in their syntactic and semantic data representations for event categories and/or event contents. For example, on one platform the channel *Temperature* might be the equivalent for the channel *Climate* on another platform. Furthermore, they can even use different *notification schemes*, e.g., channels or subjects. Last, *delivery modes* might vary among platforms for consumers and providers, i.e., push or pull.

Existing literature tries to organise interoperability issues. Some approaches are in the form of level models (e.g., [91], [129], [158], or [163]), others are classifications (e.g., [21] or [96]). Especially the approaches from [96] and [158] are interesting here because they are specifically targeting pervasive computing systems. Lahmar, Mukhtar, and Belaïd propose a classification into three categories [96]: network, protocol, and service. While this classification seems very appropriate, their descriptions are partially incomplete: The network category originally only covers the technology part of communication heterogeneity, the protocol category only addresses service discovery protocols, and the service category only takes data heterogeneity into account. On the other hand, Strang and Linnhoff-Popien focus on service interoperability, and they are the only ones (out of the mentioned) that propose an explicit context level [158]. In accordance with [115], non-functional properties also include context information. Notably, none of the approaches consider notification heterogeneity. For a complete taxonomy, the identified heterogeneities are synthesised with the classification from [96]. The three categories from [96] are employed and the seven identified heterogeneities are added as sub-categories. There, communication heterogeneity falls into the network category. Service discovery and interaction heterogeneities are included in the protocol category. The service category consists of data, application, non-functional properties, and notification heterogeneities. Figure 3.1 summarises the resulting taxonomy of heterogeneities between different platforms.

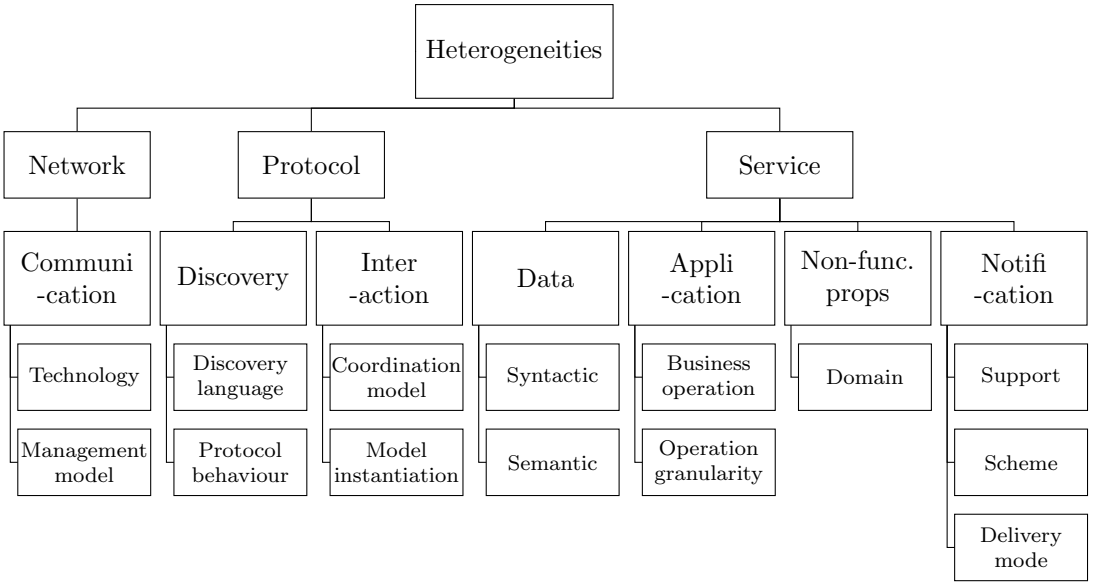


Figure 3.1.: Taxonomy of Heterogeneities. Several heterogeneities exist between different middleware platforms. According to [96], they can be classified into three categories: network, protocol, and service. (Non-func. props - non-functional properties)

Now that the difficulties that need to be overcome for achieving interoperability between different platforms are established, the next section builds a categorisation for interoperability solutions.

3.1.2. Categorisation of Solutions

Different classifications of the design of interoperability solutions have been proposed, e.g., in [21], [30], and [118]. These concepts are presented in the following. Then, these classifications are aggregated into one taxonomy. Henceforth, the term *language* is used for abbreviating communication means, data formats, protocols, and vocabulary used by a specific middleware platform.

The classification in [21] identifies the following types of interoperability solutions: traditional middleware, logical mobility, interoperability platform, software bridge, transparent interoperability, and semantic interoperability. The purpose of *traditional middleware* is to overcome heterogeneities between different systems. By using the same middleware platform, heterogeneous systems are able to communicate. Here, it is agreed upon a language in advance. However, if

different middleware platforms are used, this solution does not work any more. Indeed, these middleware platforms are the reason for the interoperability problem, and thus are the subjects to investigate and make interoperable in this thesis. Figure 3.2 depicts this approach. Examples for it are BASE [12], iPOJO [50], Limone [60], and UPnP [123].

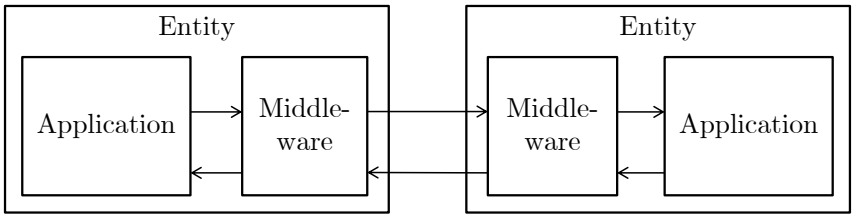


Figure 3.2.: Traditional Middleware (cf. [21]). Traditional middleware provides interoperability through agreement of the same language in advance.

In the logical mobility approach, after discovery, an entity downloads the service software and can then use it. Thus, entities are not required to know implementation details beforehand. However, entities have to agree on one common platform for code execution. Figure 3.3 shows the approach. An example for this is Jini [5].

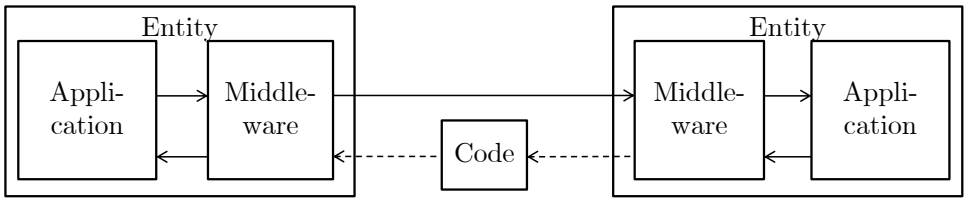


Figure 3.3.: Logical Mobility. Logical mobility is achieved through transfer of mobile code for execution (cf. [21]).

Interoperability platforms are middleware platforms that offer an application programming interface (API) for the purpose of application development and, further, provide a local mechanism that translates between the native language and other middleware languages. Thus, applications that are developed with the interoperability platform can interact with entities of other middleware platforms. However, existing applications have to be re-implemented upon the interoperability platform in order to benefit from services of other platforms. Figure 3.4 shows this approach graphically. Examples are ReMMoC [71] and MUSDAC [142].

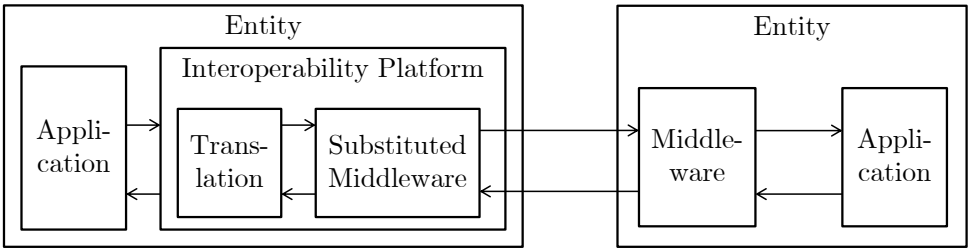


Figure 3.4.: Interoperability Platform (cf. [21]). Interoperability platforms offer an API for application development and a mechanism to translate between languages.

Software bridges enable different middleware platforms to communicate by introducing a bridge - an additional intermediate entity that translates between languages. Platforms have to be aware of the bridge to interact with it. Thus, they need to know the bridge's API. In the basic version, messages are directly translated between languages. The enterprise service bus (ESB) and enterprise application integration (EAI) are special forms of a software bridge. The difference is that they employ an indirect transformation model [79], i.e., a message is translated into an intermediate language before translating it into the target language (cf. direct versus indirect translation below). Figure 3.5 summarises this solution. Examples are IBM WebSphere Message Broker[1] and Cilia [66].

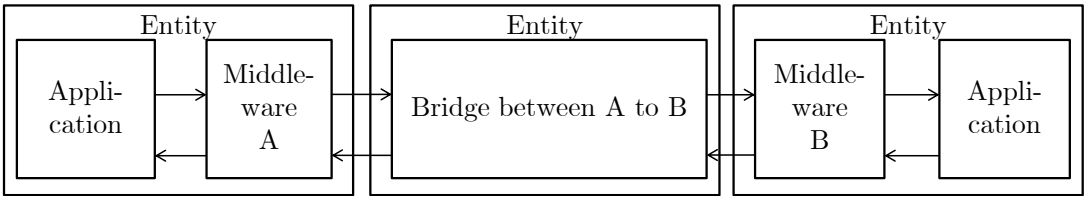


Figure 3.5.: Software Bridge (cf. [21]). Software bridges introduce an intermediate entity that translates between languages. The intermediate entity must be known by applications.

Transparent interoperability solutions use an intermediate language and format, like ESBs, to translate between languages. However, unlike with ESBs, platforms are not aware of any mediating entity. Thus, existing applications and platforms do not require any change. Figure 3.6 depicts this approach. Examples are uMiddle [118] and SeDiM [59].

[1] https://www-01.ibm.com/software/integration/ibm-integration-bus/library/message-broker/

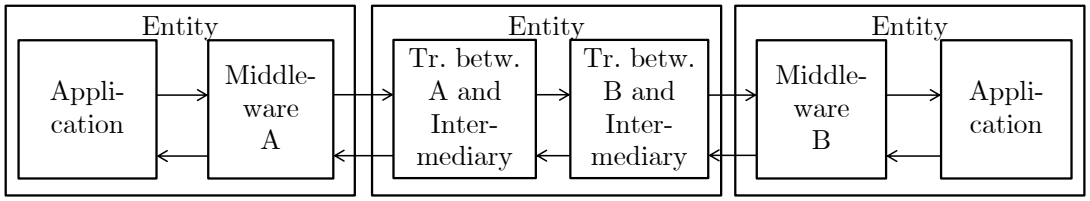
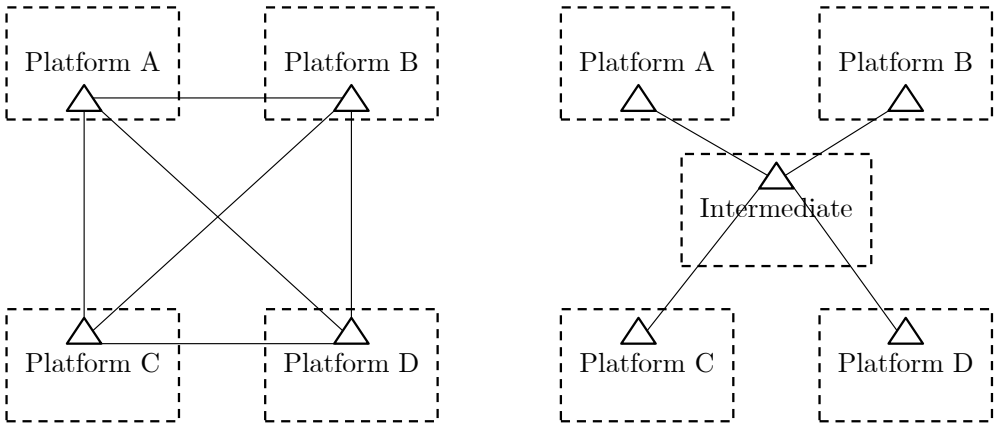


Figure 3.6.: Transparent Interoperability (cf. [21]). Transparent interoperability approaches use an intermediate entity and language to translate between languages. The intermediate entity works transparently for applications (Tr. - translation).

Semantic interoperability solutions make use of ontologies in order to provide the same language between platforms. Either platforms are forced to use the same ontology, or complex ontology alignment [46] or ontology matching [52] mechanisms are employed that translate between ontologies. Semantic interoperability specifically deals with data heterogeneity.

The categorisation by Bromberg *et al.* is broader and classifies interoperability approaches according to the awareness of the solution by entities of different platforms into explicit and transparent approaches [30]. In the *transparent* approach, messages are translated and forwarded to the communication partner without being aware of any intermediate entity or translation routine. Hence, this class differs from the transparent interoperability approach in the classification in [21] in that it potentially works with direct and indirect translation. Depending on the design of the intermediate entity, the integration of new platforms can be challenging. Nonetheless, existing platforms and entities do not demand modifications. In the *explicit* approach, an interoperability solution offers an API to entities, granting a uniform protocol. Hence, there is no need for protocol alignment. It also allows to extend existing protocols with further aspects [142], e.g., context management. Required translations confine themselves to transformations on the service level, i.e., data, application, non-functional properties, and notifications. The integration of a new platform is simple because the intermediate entity does not require any changes, only the new platform has to conform to the API. Hence, this category summarises approaches of interoperability platforms and software bridges in the classification in [21].

Furthermore, Nakazawa *et al.* propose a categorisation of interoperability solutions by the following dimensions [118]: translation model, distribution, gran-



(a) Direct Translation (b) Indirect Translation

Figure 3.7.: Translation Models (cf. [92]). In the direct translation, messages are directly translated between two platforms. In the indirect translation, intermediate semantics are used in order to decrease the number of transformers per service.

ularity, and deployment location. With respect to the *translation model*, two different designs are possible [118]: direct and mediated (also named indirect). *Direct* translation transforms a message of one platform directly into a message of another platform. The loss of information due to the translation process is minimal here because the greatest common divisor of information is translated. However, each service pair requires a transformer. Consequently, for s services and p platforms, this results in $s \cdot \sum_{i=1}^{p}(i-1)$ transformers. In other words, for adding support for one new service, $\sum_{i=1}^{p}(i-1)$ new transformers have to be integrated. Usually, the software developers specify the transformers. This can rapidly become unmaintainable when using the direct model. Figure 3.7a depicts this with an example of one service and four platforms. There, for every new service, 6 $(= \sum_{i=1}^{4}(i-1) = 0+1+2+3)$ new transformers are required. *Indirect* translation, on the other hand, uses intermediate semantics. During the translation process, it is first transformed from a message in the source format to an intermediate representation, and then from the intermediate one to a message in the target format. Here, transformers are only required between a service and the intermediate format. Consequently, having s services and p platforms, the amount of transformers decreases to $s \cdot p$. In other words, in order to add support for one new service, p new transformers have to be integrated. This

results in equal or less transformers for the indirect translation model for $p \geq 3$. Furthermore, this lowers the hurdle for adding new platforms and services, and consequently increases extensibility. However, information might get lost as the intermediate format might contain less information than the greatest common divisor between only two platforms [21]. Figure 3.7b shows this with an example of one service and four platforms. For every new service, 4 new transformers are required there.

Distribution is concerned with the visibility of translated services [118]. Using a *scattered* approach, every service is made visible to any platform using proxy representations. Thus, every entity can seamlessly access other services. The *aggregated* model does not distribute services to other platforms, but stores them in an intermediate format. Only entities that are implemented with the intermediate platform can access those services, not being visible in other platforms. The direct translation model implies a scattered visibility, whereas in the indirect model both solutions, scattered and aggregated, are possible.

The *granularity* dimension is concerned with the representation of services [118]. A *coarse-grained* representation includes service types and operations. If service types coincide, services are considered to be compatible (syntactic matching). Using this approach, only services that are currently defined can be translated and accessed. A *fine-grained* representation includes service semantics, i.e., provided and required inputs and outputs together with data types. If a consumer's provided input and required output matches a service's required input and provided output, they are compatible (semantic matching). This dimension is specific to the indirect translation.

Considering the *deployment location*, an interoperability solution might be located in the infrastructure or at-the-edge [118]. If the interoperability solution is located on the devices themselves, it is referred to *at-the-edge*. In contrast to that, it is called *in the infrastructure* if the solution is deployed on a dedicated node, such as a router or server, possibly in the cloud [4]. The latter approach usually does not require changes at existing devices or services, whereas the former one does. Usually, at-the-edge solutions communicate without any intermediate node, whereas solutions deployed in the infrastructure, as one can suggest, are deployed on an intermediate entity.

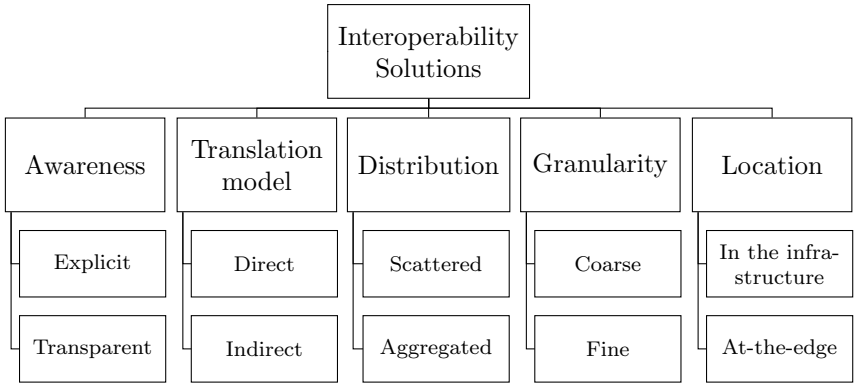


Figure 3.8.: Taxonomy of Interoperability Solutions. Interoperability solutions can be classified by several categories.

Figure 3.8 summarises the different categories in one taxonomy with the following dimensions: awareness, translation model, distribution, granularity, location, and discovery model. The classification in [21] is omitted because some types directly imply other design decisions. Furthermore, the differentiation in [30] covers a similar aspect but is more general. The next section introduces general requirements for an interoperability framework.

3.1.3. Requirements for an Interoperability Framework

Several requirements have been identified for interoperability frameworks for pervasive computing systems. The identified requirements are, on the one hand, based on the purpose of a framework and, on the other hand, derived from characteristics of pervasive computing systems. In general, a framework is an abstraction that provides generic functionality and can be extended by user-written code. Frameworks usually offer re-usable and customisable components in order to build application-specific software. The requirements are non-exhaustive.

R1 **Extensibility**: There already exists a high variety of different middleware platforms for pervasive computing and IoT and most likely more will be developed. Furthermore, for each platform, new services will be implemented in future. That is why an interoperability solution must be extensible with new platforms and new services [118].

R2 **Customisability**: Depending on the domain (or also due to future evolution of middleware platforms), simpler or more sophisticated mechanisms

(e.g., for message translation) might be desired. These mechanisms and/or parts of them should be easily customisable, exchangeable, or extensible. Therefore, this requirement demands explicit support for the developer in order to customise internal algorithms.

R3 **Dynamism**: In pervasive environments, there co-exist many mobile devices. Those can be also of mobile nature, such as smartphones, and hence change their location regularly [21]. The wide usage of the SOC paradigm enables those entities to join and leave the system at runtime. An interoperability solution must be able to cope with this dynamism introduced by the fluctuation of devices.

The next section introduces related approaches that aim at interoperability between pervasive computing and IoT platforms, and discusses them with respect to the elaborated evaluation framework.

3.2. Analysis of Existing Approaches

Literature has presented several reviews regarding interoperability between different middleware platforms. Some literature reviews exist especially with respect to service discovery heterogeneity in mobile or pervasive computing (e.g., [1], [39], [77], [103], [111], [143], [144], and [173]). The review by Zhu *et al.* [173] is the most complete one, dividing a service discovery protocol in ten different components: service and attribute naming, initial communication method, discovery and registration, service discovery infrastructure, service information state, discovery scope, service selection, service invocation, service usage, and service status inquiry. The most commonly identified differences in those reviews is the service discovery infrastructure (or also architecture [39]), i.e., directory-based versus directory-less. Other reviews do not only consider service discovery heterogeneity but also include other heterogeneities. Blair *et al.* show a non-exhaustive overview on interoperability solutions for complex distributed systems and classify them according to a set of heterogeneities [21]. There, complex distributed systems include grid applications, mobile *ad hoc* applications, enterprise systems, and sensor networks. Lahmar [96] presents a non-exhaustive overview on interoperability approaches for pervasive computing categorised by

service interoperability, service discovery protocol interoperability, and network interoperability. In [112], IoT platforms are classified according to some criteria including their support for heterogeneous devices. Based on the classification, the authors give recommendations on how to extend each platform in order to meet the criteria. For more details, the interested reader is referred to [21], [96], and [113], respectively.

The remainder of this section is partially based on these overviews but gives a more up-to-date version, also including IoT approaches. Furthermore, the approaches are evaluated against the above introduced evaluation framework (see Section 3.1). Each approach is described and assessed one by one. A differentiation is made between approaches targeting interoperability in pervasive computing environments (see Section 3.2.1) and approaches targeting interoperability in IoT environments (see Section 3.2.2). If information on certain criteria are not known about an approach or not identifiable from literature, it is not explicitly mentioned.

3.2.1. Pervasive Computing Approaches

In the following, related work approaches with respect to pervasive computing are presented. This includes pervasive computing, context-aware computing, mobile computing, and ambient intelligence approaches.

Speakeasy [45] is an interoperability platform. It uses a fixed set of generic interfaces for data transfer, collection, metadata, and control, in order to avoid making interfaces between heterogeneous services conform. Custom objects that implement one of these known interfaces are used as return types, and are then used to access services. Unknown interfaces, data types, as well as protocols can be loaded through mobile code. As a result, prior knowledge on protocols or data types is not necessary. Furthermore, applications and services only have limited knowledge due to the small set of generic interfaces. Thus, the user is the one who decides when and how to access remote services as the authors assume that users understand the service-specific semantics intended by developers. This prevents applications from an automatic execution. Developers have to create bridges that allow Speakeasy applications to discover and access services. However, existing applications cannot make use of services running on other platforms. Speakeasy

fulfils partially service discovery heterogeneity (H2) and syntactic data heterogeneity (H4). The authors mention that notifications can be received, but it is unclear if any transformations happen (H7). Moreover, Speakeasy uses an explicit interoperability access, indirect translation with aggregated visibility, and it is deployed at-the-edge. Further, it fulfils the platform extensibility (R1) and dynamism (R3) criteria.

The approach in [62], hereafter called *SQL Broker*, is an interoperability attempt for service discovery. It uses a shared data space, service interpreters and a database. A service interpreter must exist for each supported service discovery protocol. It monitors the network for service advertisements, translates them into structured query language (SQL) INSERT statements, and writes them to the shared data space. The database monitors the shared data space, takes out SQL statements, and executes them. It stays unclear how translation of service descriptions, if performed, happens. SQL Broker allows to integrate existing services without changes to those. Applications, however, need to use a specified API in order to query for available services. Furthermore, SQL Broker only solves a part of the service discovery heterogeneity (H2) because some points remain unclear, e.g., translation of service descriptions. The approach is explicit and performs indirect, scattered translation. It is deployed in the infrastructure. Service interpreters allow to extend the SQL Broker with new platforms (R1). The dynamism (R3) criterion is partially tackled as it covers services entering and leaving the system explicitly.

In [3], the authors present a transparent interoperability approach between UPnP [123] and Jini [5], here called *Jini/UPnP*. They use proxies in order to enable clients in UPnP to use Jini services and the other way around. By this, Jini clients can discover UPnP services and, respectively, UPnP clients can discover Jini services. Furthermore, these proxies transform access calls between Jini and UPnP and interact with the corresponding service. A 'modest amount of code' [3] is required for each of these proxies which incorporate one service type. For each direction, UPnP to Jini and vice versa, a separate module is needed that creates proxies. Jini/UPnP tackles service discovery heterogeneity (H2), syntactic data heterogeneity (H4), and translates between operation names and parameters (H5). Additionally, it bases on a transparent awareness approach, employs direct translation with scattered distribution, and is deployed in the

infrastructure. Also, it fulfils the extensibility (R1) criterion, with respect to services, as well as the dynamism (R3) criterion.

Koponen and Virtanen present in [92] a transparent interoperability approach, henceforth called *Broker*, to allow for service discovery interoperability between Jini [5] and SLP [75]. It employs a broker engine and, for each supported middleware platform, an adapter. The broker engine acts as message mediator that forwards translated service registrations between adapters. Adapters makes use of transformers in order to translate service registration messages into an abstract format and representation before passing the messages to the broker engine. A transformer must exist for every supported service type. Service requests do not require translations as adapters pass them to either a local lookup service or directory agent. Service access is not considered in the Broker approach. Besides addressing service discovery (H2) and data (H4) heterogeneities, the Broker approach is able to send notifications between Jini and SLP (H7). However, it remains unclear how inter-platform notifications work. The Broker approach works transparently, uses indirect transformation, scattered distribution, coarse-grained granularity, and is deployed in the infrastructure. Furthermore, new platforms can be integrated by adding further adapters, and new services can be added through new transformers, which requires writing code (R1). Dynamism is handled as well (R3).

The reflective middleware to support mobile client interoperability (*ReMMoC*) [71, 72] is an interoperability platform for mobile environments based on reflection and a component framework. ReMMoC is implemented based on the OpenCOM component framework [41] and includes a mapping component as well as further component frameworks for service discovery and binding. The web service description language (WSDL) [40] is used to describe services of different middleware platforms. The mapping component uses WSDL files for mapping between abstract services and concrete bindings, and vice versa. The service discovery and the binding component frameworks need to include components for supported protocols, e.g., remote method invocation (RMI) using SOAP [26]. During runtime, reflection may lead to component replacements. ReMMoC offers an API for developing mobile applications that can interact with services using different middleware platforms. It does not provide interoperability between existing services. Only a mobile client that is implemented with ReMMoC

can access services of other platforms. This requires tremendous changes (re-implementation with ReMMoC) to existing applications if they want to make use of ReMMoC. ReMMoC addresses service discovery heterogeneity (H2) and interaction heterogeneity (H3). It is an explicit approach that uses indirect, aggregated translation, and is deployed at-the-edge. ReMMoC is extensible with new platforms and services (R1), and copes with a dynamic environment (R3).

The universal middleware bridge (*UMB*) [117] aims at interoperability for home network middleware platforms. It consists of a core and several adaptors. Adaptors abstract middleware-specific devices to the universal device template, the common device representation employed in UMB. The abstraction happens with mappers. The abstract device representation is then passed to the core, where it is stored. Core and adaptors are connected through the network. Universal metadata messages, which the core routes between the adaptors, allow for lookup and interaction between heterogeneous devices. Addressed heterogeneities by UMB are service discovery (H2), data (H4), and partially application (H5) since operation names and parameters are translated. UMB is a transparent approach using indirect, scattered, coarse-grained translation, and is deployed in the infrastructure. It is extensible with new platforms through adding adaptors, and with new services by adding new device descriptions (R1). As well, it handles dynamism (R3).

The multi-protocol service discovery and access middleware platform (*MUSDAC* [142] or *MSDA* [141]) allows for service discovery and access across pervasive middleware platforms. MUSDAC uses an explicit interoperability approach where it registers itself as service at each platform. Applications can use MUSDAC through the provided API and the MUSDAC representation for service discovery and description in order to discover, lookup, and access services. For each supported platform, a plugin must be present that bears responsibility for performing local service discovery and access tasks. Within plugins, transformers extend service descriptions with context information. Managers forward requests between plugins and/or bridges. Bridges are gateways to other networks where a MUSDAC instance is present. Clients may restrict the dissemination of certain requests to specific contexts, e.g., the local network. As others, this approach makes use of an intermediate message format. For this purpose, the authors introduce the abstract MUSDAC description format for service descriptions, as

well as the abstract MUSDAC request format for service requests. MUSDAC's explicit approach indeed enables extension of service descriptions. Though, it requires substantial changes in existing platforms. Furthermore, because clients already use the abstract formats during communication, transformations due to heterogeneous data formats is not required. It is unclear if vocabulary is translated, or a common vocabulary is assumed. MUSDAC copes with communication heterogeneity (H1) and service discovery heterogeneity (H2). Data (H4) and non-functional properties (H6) heterogeneities are partially met because it remains unclear if vocabulary translation happens. It employs an indirect, scattered translation model and is deployed in the infrastructure. Further, it is extensible with respect to platforms and services (R1), and handles dynamism (R3).

The pervasive semantic syntactic (*PerSeSyn*) service discovery platform [17], as part of the PLASTIC framework [116], aims at interoperable service discovery for ambient computing environments. The communication middleware of the PLASTIC framework bases on MUSDAC. Hence, managers coordinate the discovery in one independent network, and are able to communicate with managers from other networks via bridges. Additional technical services are implemented on top, including a service accessibility and composition service of which the PerSeSyn platform is a part. The PerSeSyn platform allows for service discovery between syntactic-based and semantic-based service discovery protocols. It especially focuses on an abstract service specification and on an algorithm for service matching. The authors propose the PerSeSyn service specification consisting of an abstract model and language. It is based on semantic annotations for WSDL (SAWSDL) [54] and the web services business process execution language (WS-BPEL) [121]. It provides support for syntactic as well as semantic service descriptions. For service matching, depending on the descriptions of a request and the service to be checked, a different algorithm may be used, e.g., if both request and service are described syntactically, a syntactic matching is performed. The semantic matching algorithm, which is performed if request and service are described by semantic capabilities, is based on [115]. Due to the fact that MUSDAC builds the communication layer, clients can benefit of the PLASTIC framework (and thus of the PerSeSyn platform) by using the provided API. With respect to the evaluation framework, PerSeSyn covers the same aspects as MUSDAC.

The interoperable discovery system for networked services (*INDISS*) [28, 29] provides transparent service discovery interoperability through the use of event-based parsing methods. A monitor component maps messages to a platform based on the port and address on which the message is received. The corresponding parser transforms the message into a series of events and passes those to the responsible composer. The composer then converts the events into a message that is understood by the target device. A set of mandatory events for service discovery are proposed. As some platforms offer more functionality than others, platform-specific events can be integrated. If a composer does not understand certain events, they are discarded. Parsers and composers can be switched at runtime, e.g., from an HTTP parser to a SOAP parser, to support platforms that are based on several protocols, such as UPnP [123]. The coordination of events in parsers and composers of a platform works with the help of finite state machines. INDISS addresses service discovery (H2) heterogeneity, neglecting service access. Yet, it assumes a common vocabulary as no translations between message content is mentioned. Further, INDISS is transparent, and it performs translations directly. It is deployable in the infrastructure as well as at-the-edge. Also, it is extensible with platforms and services (R1), and manages dynamism (R3).

The *Amigo* interoperability framework [69, 162] targets interoperability for ambient intelligence in the networked home. It bases on INDISS as interoperable service discovery mechanism and also takes over the event-based parser concept for enabling interoperable service interaction. Service matching happens according to PerSeSyn [17]. Discovered service descriptions are used to generate stubs that bridge service invocations between a client and a service. A stub consists of a parser and a composer unit (cf. INDISS [28, 29]) that generate semantic events upon message arrival. Furthermore, on the communication protocol level, Amigo offers 'per-layer interoperability' [69], i.e., each layer of the protocol is translated if the target protocol is sufficiently similar to the source protocol. Thus, for each layer there has to be a parser and a composer present. Moreover, the Amigo service repository offers an API that allows clients to lookup services directly. For this, an alteration of existing client applications is required. Regarding heterogeneities, Amigo tackles communication heterogeneity (H1), service discovery heterogeneity (H2), and partially application heterogeneity (H5) because only operation names and parameters are aligned. Amigo is a transparent solution that

uses direct translation. Besides, it is deployable in the infrastructure as well as at-the-edge. Additionally, it is extensible with new platforms and services (R1), and addresses the dynamism (R3).

Also as part of the Amigo project [69], Uribarren *et al.* [164] propose a transparent interoperability approach, hereafter called *Bridge*, that allows applications to use hardware services, like wireless sensors or RFID locators. It consists of a drivers layer, a unified model layer, and a bridges layer. In the drivers layer, drivers communicate with devices, extract service descriptions from available devices, and generate unified descriptions from those. Each device technology requires a driver. The unified model layer is, basically, a service registry that stores the available services in the unified description format. The unified description of a service includes information on methods, properties, events, and meta-information. In the bridges layer, for each supported platform, a service instance is created from a unified service description. Existing applications can then discover and access the service instances. Unfortunately, the authors do not give any specifics on required translations. The approach tackles communication heterogeneity (H1) and service discovery heterogeneity (H2). It is transparent and uses indirect translation with scattered visibility and coarse-grained granularity. It targets a deployment in the infrastructure. Further, the approach is extensible (R1) and supports dynamism (R3).

uMiddle [118] is a bridging framework to enable interoperability between middleware platforms. Therefore, it uses an intermediate representation, mappers, and translators. For the intermediate representation, the authors propose the universal service description language [118, 119] (USDL). USDL is based on XML and consists of a platform-independent and a platform-dependent part. Mappers discover services of one middleware platform using the platform-specific service discovery protocol, and import them into the uMiddle registry. They automatically generate USDL files for discovered services. Unfortunately, it is not clear how this works from the papers. Translators bridge from a platform-specific representation into the intermediate uMiddle representation, and vice versa. Moreover, the uMiddle framework provides an API for developing platform-independent applications. Only applications that have been developed using the API can access services of different platforms because discovered services are not propagated to other platforms. Thus, interoperability between services is only offered when

using a uMiddle application. uMiddle addresses communication (H1), service discovery (H2), and syntactic data (H4) heterogeneities. It works transparently, uses indirect translation with aggregated visibility and a fine-grained representation, and is deployed in the infrastructure. Moreover, uMiddle covers extensibility (R1) only partially because clients that are not implemented with uMiddle cannot benefit from other platforms. Additionally, it deals with dynamism (R3).

Flores *et al.* present a multi-protocol framework for *ad hoc* service discovery (*MPFSD*) [58, 59]. They distinguish between directory-based and directory-less service discovery protocols (like, e.g., [39], [143], and [173]) and establish common interaction patterns for both. Based on this, they identify common components among several service discovery protocols, such as advertiser, request, reply, network, cache, and policy. For integrating new protocols, these components need to be implemented. However, this solution neglects the fact that different service discovery protocols use different service description languages, and hence does not employ any translation mechanism. Thus, it only enables multi-protocol service discovery if the same service description language is spoken. Therefore, MPFSD only tackles part of service discovery heterogeneity (H2) as no translations take place. It uses a transparent approach. Message translation happens indirectly for scattered visibility. Moreover, MPFSD is deployed in the infrastructure. The approach can be extended with new platforms (R1).

SeDiM [57] is a configurable transparent interoperability framework for enabling service discovery across platforms and also for developing applications with the provided API. It is presumably based on the approach of [58, 59], and extends it with a discovery event abstraction, a domain hub, and a service description abstraction. The discovery event abstraction works as intermediate message representation. Thus, messages are not translated one-to-one, but indirectly via the abstraction. The domain hub is responsible for translating messages between different protocol formats. For every supported platform, there has to be a domain socket that is connected to the domain hub. Every message received in a domain socket is forwarded to the domain hub, translated, and forwarded to the target domain socket. The service description abstraction enables service lookup and matching among different middleware platforms. Therefore, they use a format based on WSDL. Furthermore, according to [57], SeDiM can be configured dynamically based on the current environment, i.e., if a certain middleware plat-

form is detected, it is configured. Instances of SeDiM are able to communicate with each other if both instances overlap in one domain socket. Unfortunately, the paper is on a very high level and no detailed information about its internal functioning is provided. SeDiM addresses service discovery heterogeneity (H2). It employs a transparent approach, indirect and scattered translation, as well as deployment in the infrastructure and at-the-edge. Further, the approach supports extensibility with new platforms and services (R1), and dynamism (R3).

The open service discovery architecture (*OSDA*) [106] enables transparent cross-domain service discovery through a distributed information storage and querying model, and an abstract information representation. The authors define a domain as 'a federation of network components [...] controlled by a single service discovery technology' [106]. Service brokers bridge between intra-domain and inter-domain discovery systems. Therefore, service brokers consist of two layers. The lower layer intercepts messages of the local domain and translates them to the abstract representation, and vice versa. The upper layer manages communication on the inter-domain level, i.e., the communication between service brokers. Here, a structured peer-to-peer overlay network, based on a distributed hash table, is used to advertise services and process service queries. For an abstract information representation, the authors propose the Unified Service Description scheme as service description format, as well as the Unified Request and Unified Response schemes for service requests and, respectively, service responses. Their proposed schemes base on XML. OSDA supports service advertisement and querying, but no access. Unfortunately, the authors do only provide few information on the translation mechanism. OSDA tackles service discovery heterogeneity (H2). The approach is transparent, and it uses indirect, scattered translation. It is deployed in the infrastructure. Moreover, new platforms and services can be integrated through new service brokers (R1), and dynamism is addressed (R3).

Smart-M3 [80] is a platform for information sharing among heterogeneous devices. The approach includes semantic information brokers, knowledge processors, and the smart space access protocol. Semantic information brokers store information of a smart space in a database, i.e., context information. They further must implement operations provided by the smart space access protocol. Then, knowledge processors can join/leave semantic information brokers, insert/remove/update/query information, and subscribe/unsubscribe for certain

changes in the information. Communication between semantic information brokers and knowledge processors is not fixed, i.e., different semantic information brokers can use different transport protocols. Knowledge processors further implement an application logic based on a chosen ontology. Information interoperability is actually achieved through ontology standardisation. Hence, Smart-M3 addresses communication (H1) and data (H4) heterogeneity. It is an explicit approach using indirect transformation, and can be deployed in the infrastructure. Because Smart-M3 provides data rather than service interoperability, the distribution and granularity classification is not feasible. It tackles extensibility (R1) and dynamism (R3) requirements.

Kiljander *et al.* extend Smart-M3 with ucodes (based on the Ubiqitous ID architecture [93]) to find object information on a world-wide scale [91]. This approach is referred to as *Ubi-M3* hereafter. Virtual entities, representing devices, and semantic information brokers are identified by ucodes. Entities can communicate with a resolution infrastructure in order to discover and look other entities up based on ucodes. The approach uses the same concepts as Smart-M3.

With *InterX* [132], the authors propose an interoperability gateway for pervasive computing devices that runs on smartphones. There, one smartphone represents one user having several devices with different service protocols. They assume that service protocols are not known beforehand. Thus, first of all, a service protocol discovery takes place where InterX instances exchange a list of their supported service protocols. For non-overlapping protocols, a component is created that translates service discovery requests and responses. A service proxy is then instantiated for every discovered service. Translations happen based on a common abstraction format. The authors underline that interoperability is enabled at runtime. However, no information is given on how service discovery components and service proxies are instantiated and configured at runtime. InterX requires that devices use the API for exchanging their supported protocols. Only service discovery (H2) heterogeneity is addressed and happens on a transparent basis. Furthermore, InterX performs indirect, scattered transformation, and is deployed at-the-edge. It is extensible (R1), and it handles dynamism (R3).

ZigZag [145] is a middleware for service discovery among different networks. It consists of four components: monitor, connector management, network link, and aggregator. The monitor component, as introduced by INDISS [28, 29], de-

tects available service discovery protocols depending on the multicast group address and port. Each service gets assigned to a universally unique identifier [102] (UUID) in order to identify a service globally. For each pair of service discovery protocols, the connector management component has to instantiate a connector. Thus, ZigZag uses a direct translation mechanism. The network link component maintains an overlay network between ZigZag instances for exchanging requests and services. Because potentially many responses arrive for one request and, further, existing service discovery protocols are not designed to handle such a great amount of messages, the aggregator component aggregates these responses, selects the best matching response, and returns it to the requester. ZigZag is deployed in the infrastructure, and focuses on service discovery (H2) heterogeneity. It is a transparent solution that uses a direct translation model. Also, it is extensible with new platforms and services (R1), and supports the dynamism (R3) criterion.

As part of the CHOReOS project [167], the extensible service bus (*XSB*) [68] is developed. The XSB bases on an ESB and targets interaction paradigm interoperability. Middleware platforms are abstracted as connector models. The authors provide connector models for the client-server, publish-subscribe, and tuple space paradigms. The specific connector models are further abstracted as a generic connector model. Binding components can convert between the different representations using protocol conversion techniques [100]. Furthermore, the XSB approach uses a proprietary interface description language (IDL) to describe deployed services. Binding components are automatically created from the service descriptions. The approach tackles interaction (H3) heterogeneity. Besides, it is explicit, uses indirect translation, and is deployed in the infrastructure. Regarding the requirements, it is extensible with new platform and services (R1).

In the *CONNECT* project [15, 16], software connectors are the basis for interoperability. CONNECT shifts all activities to runtime, not requiring any domain-specific knowledge at design time. Connectors are automatically generated at runtime if two entities spontaneously decide to interact. This happens with the help of learning and synthesis techniques [18]. Learning techniques are used to derive ontologies for services as well as behavioural and semantic specifications of applications and services [13]. Ontologies are learnt through text categorisation with support vector machines. For behavioural and semantic specifications,

CONNECT applies an iterative process in which interactions are actively tested and refined. In case two systems want to interact, their specifications are synthesised during connector generation. Due to the usage of learning techniques, erroneous categorisations can conceivably happen. All in all, CONNECT addresses service discovery (H2), interaction (H3), data (H4), and application (H5) heterogeneities. Moreover, it is a transparent approach performing direct translations, and can be deployed in the infrastructure. CONNECT is extensible (R1) and handles dynamism (R3).

3.2.2. Internet of Things Approaches

In the following, related work approaches with respect to the Internet of Things are presented. This includes Internet of Things and Web of Things approaches.

Dynamix [34, 35] is a framework for the Web of Things. It allows Web-based applications to discover and access non-Web-based services (as, e.g., pervasive computing devices). Dynamix is designed to be executed on smartphones running the Android[2] operating system. Web-based applications (e.g., Web browser or Android applications) may want to interact with non-Web-based services. Therefore, an Android service provides representational state transfer (REST) APIs for Web applications and Android IDL (AIDL) APIs for Android applications. Abstract protocols have to be created that conform to protocol-specific implementations. Web-based applications then can send abstract invocations which the Android service translates to protocol-specific invocations. Furthermore, Dynamix provides an API for implementing Web-based applications in order to communicate with the Dynamix (Android) service. It is not designed for interoperability between existing services, but only to access services from Web-based applications that implement the Dynamix API. The Dynamix approach tackles service discovery heterogeneity (H2). It works explicitly, uses indirect, aggregated transformation, and is deployed at-the-edge. Further, Dynamix is extensible (R1) and handles dynamism (R3).

oneM2M [94, 160] aims at providing semantic interoperability between machine-to-machine solutions through standardisation, i.e., by developing a global specification. It bases on FIWARE [139] as service discovery framework. FIWARE

[2]https://www.android.com/

is a framework using generic enablers. Those are easily configurable and deployable cloud components. oneM2M builds a layer on top of FIWARE in order to integrate semantic reasoning and knowledge processing to allow for data interoperability. Semantic mediation gateways are placed between these two layers and are responsible for transforming data representations based on semantic annotations. For translation purposes oneM2M employs device and resource abstractions. Further, it provides a common API in order to develop new services. Interoperability with existing services is mentioned [160], but it stays unclear how it works. Therefore, oneM2M only addresses data heterogeneity (H4). The approach is presumably explicit, and uses an indirect, scattered translation scheme. It is deployed in the infrastructure.

HyperCat [20, 82] refers to itself as 'IoT interoperability specification' [82] with the goal of having one shared representation and query mechanism for IoT resources. It specifies a hypermedia catalogue format and an API to expose and query information on IoT resources over the Internet. Service information, or more specifically the hypermedia catalogue, is stored in so-called hubs and is represented as statements based on the Resource Description Framework (RDF) [101]. The API allows to expose services to hubs, and thus, if existing services should be included, they need to be adjusted to conform with the HyperCat API. Deletion of information, e.g., in case of leaving/shutting down, takes place explicitly. HyperCat is an explicit solution that works with indirect translation. As the approach seeks for data interoperability, the distribution and granularity classes are not feasible. Instances are deployed in the infrastructure.

The *IoT Hub* [113] approach provides interoperability by extending existing middleware platforms with a REST API for communication with an IoT hub. So-called enablers bridge to devices by using the REST API and so-called IoT feeds. Things and data are abstracted to such IoT feeds using transformations. IoT feeds are stored in the IoT hubs that also provide an API for storing and accessing them. In order to enable interoperability between different IoT hubs, there exist meta hubs. Meta hubs store information about available services and IoT hubs. Furthermore, an API is provided for building applications that can use available services. The approach tackles data heterogeneity (H4). It works explicitly, employs an indirect approach, and is deployable in the infrastructure. Again, data interoperability is to the fore, and thus, distribution and granularity

is not considered. Further, IoT Hub covers the extensibility (R1) and dynamism (R3) criteria.

The IoT European Platforms Initiative[3] (IoT-EPI) is an initiative for IoT platform development. It aims at building a sustainable IoT ecosystem, and it comprises seven – still ongoing – projects of which four revolve around interoperability at the communication, protocol, or service level. Those are symbIoTe, bIoTope, BIG IoT, and Inter-IoT.

The *symbIoTe* [70, 89, 157] project tries to provide interoperability between IoT platforms through a uniform access to services. It bases on the oneM2M architecture, uses a common representation, and consists of the Core API, Interworking API, and the symbIoTe middleware. The symbIoTe middleware is responsible for service discovery and configuration. For this purpose, it offers a standardised API. Platform providers have to implement the Interworking API into their platform in order that services can be uniformly accessed. Here, platform-specific adaptors enable syntactic transformation of data. The Core API can be used by application developers for service querying. This API needs to be implemented into existing applications that they can benefit from symbIoTe. Also, the Core API semantically transforms service queries between the common and platform-specific representation. The symbIoTe approach addresses communication (H1), service discovery (H2), and data (H4) heterogeneities. Further, it presumably is explicit. It uses indirect, scattered translation, and is deployed in the infrastructure. The approach meets the extensibility (R1) and dynamism (R3) criteria.

bIoTope [63, 95] wants to use standards for enabling interoperability across IoT platforms. Therefore, it uses the Open Message Interface (O-MI) as communication abstraction and the Open Data Format (O-DF) as description format. O-MI specifies an API for RESTful communication that is not bound to HTTP. No further details are known so far. bIoTope tackles data heterogeneity (H4).

BIG IoT [27] aims at providing platform interoperability for IoT ecosystems. For achieving this, BIG IoT offers APIs and a marketplace. The marketplace acts as service registry and also stores usage information of consumers and providers for charging consumers correctly. APIs exist for the marketplace, as well as for providers and consumers. The marketplace API offers operations for several

[3] http://iot-epi.eu

functions, such as registration, discovery, and charging. The API for providers and consumers provides operations for accessing services and authenticating at the marketplace. Here, translations of service calls must happen. In order to enable interoperability with BIG IoT, existing services or platforms can either be extended with the API, or so-called gateway services can be implemented which translate between platforms and BIG IoT, or proxy services can be used that handle interactions with the marketplace [153]. BIG IoT addresses service discovery heterogeneity (H2) and data heterogeneity (H4). Further, it works explicitly and employs indirect, scattered translation, and is deployed in the infrastructure. BIG IoT is extensible (R1), and it supports dynamism (R3).

INTER-IoT [63, 65] targets a framework for an easy development of interoperable IoT devices. Desired outcomes of this project are the INTER-LAYER, the INTER-FW, and the INTER-METH artefacts. The INTER-LAYER enables interoperability between IoT platforms through virtual gateways, virtual switches, a 'super middleware' [63], a service broker, and a semantics mediator. The INTER-FW allows to access the INTER-LAYER in order to create interoperable IoT services and applications. INTER-METH is a methodology that helps with the integration of platforms. INTER-IoT covers communication (H1), service discovery (H2), data (H4), and partially application (H5) heterogeneities. It is an explicit approach that uses indirect, scattered translation. It can be deployed in the infrastructure as well as at-the-edge. Moreover, the approach is extensible (R1) and dynamic (R3).

IoTivity [124] is a framework for seamless connectivity between IoT devices based on the Open Connectivity Foundation [105, 133] (OCF) standards. The OCF tries to establish an IoT standard that is also able to bridge between IoT ecosystems, such as oneM2M or AllJoyn, and integrate devices using traditional connection technologies, e.g., ZigBee or Bluetooth. IoTivity uses these standards to enable interaction between heterogeneous, smart, and thin devices. Things and devices are abstracted as resources, and an API is provided to manipulate those resources. Addressed heterogeneities are communication (H1), service discovery (H2), and data (H4) heterogeneities. IoTivity is transparent, uses indirect, scattered translation, and is deployed in the infrastructure. Also, it is extensible (R1) and handles dynamism (R3).

As part of the CHOReVOLUTION project[4], the eVolution Service Bus (*VSB*) [23, 24] aims to overcome interaction heterogeneity between business-oriented and Things-based services. The VSB is based on an ESB and, thus, provides a common bus protocol. Binding components translate between concrete interaction protocols and the common bus protocol. This happens with a description of a service/thing based on their own generic interface description language. The description also includes a mapping for operations and data. Binding components further make use of a protocol pool which stores supported protocols. Per service/thing, one binding component is required. The VSB approach addresses interaction heterogeneity (H3) and partially application heterogeneity (H5) as operation names and parameters are translated. It works explicitly, employs indirect, scattered transformation, and is deployable in the infrastructure. Also, the VSB approach is extensible with new platforms and services (R1).

3.2.3. Summary

The introduced approaches in this section try to enable interoperability between pervasive and IoT platforms. However, none of the approaches fulfils all of the elaborated heterogeneities. Especially, interaction and notification heterogeneities are rarely addressed. Many of the IoT solutions try to specify new standards for developing IoT services and applications. Those approaches also state that interoperability with existing devices is ensured. Regarding this aspect, unfortunately, only few information can be found in the respective papers. Furthermore, many approaches focus on the semantic data heterogeneity and neglect other heterogeneities. As they introduce APIs in order to use their solutions, this is feasible. Though, existing services and applications are excluded in that way. Moreover, the goal of a framework is to support developers for a specific task. Here, this task is to achieve interoperability between platforms. Although claiming to be interoperability frameworks, only few of the presented approaches, however, offer developers freedom with respect to the solution. Developers are constrained to the frameworks' pre-defined models and cannot choose between options, e.g., deployment in the infrastructure or at-the-edge. Table 3.1 summarises the assessment of the the presented approaches.

[4]http://www.chorevolution.eu

	Project	Heterogeneities							Solution Design												Reqs		
									Aw.		Tr.		Di.		Gr.		Lo.						
		Communication	Discovery	Interaction	Data	Application	Non-func. props	Notification	Transparent	Explicit	Direct	Indirect	Scattered	Aggregated	Coarse-grained	Fine-grained	In the infrastructure	At-the-edge	Extensibility	Customisation	Dynamism
Pervasive Computing	Amigo [69, 162]	●	●			○			●		●		●				●	●	●		●
	Bridge [164]	●	●						●			●	●		●		●		●		●
	Broker [92]		●		●			○	●			●	●		●		●		●		●
	CONNECT [15, 16]		●	●	●	●			●		●		●				●		●		●
	INDISS [28, 29]		●						●		●		●				●	●	●		●
	InterX [132]		●							●		●	●					●	●		●
	Jini/UPnP [3]		●		○	○			●		●		●				●		○		●
	MPFSD [58, 59]		○						●			●	●		●		●		●		
	MUSDAC [141, 142]	●	●		○		○			●		●	●				●		●		●
	OSDA [106]		●						●			●	●		●	●	●		●		●
	PerSeSyn [17]	●	●		○		○			●		●	●		●	●	●		●		●
	ReMMoC [71, 72]		●	●						●		●		●				●	●		●
	SeDiM [57]		●						●			●	●		●		●	●	●		●
	Smart-M3 [80]	●			●					●		●					●		●		●
	Speakeasy [45]		○		○			○		●		●		●				●	●		●
	SQL Broker [62]		○							●		●	●				●		●		○
	Ubi-M3 [91]	●			●					●		●					●		●		●
	UMB [117]		●		●	○			●			●	●		●		●		●		●
	uMiddle [118]	○	○	○	○				●			●		●		●	●		●		●
	XSB [68]			●						●		●					●		●		
	ZigZag [145]		●						●		●		●		●	●	●	●	●		●
Internet of Things	BIG IoT [27]		●		●					●		●	●				●		●		●
	bIoTope [63, 95]				●																
	Dynamix [34, 35]		●							●		●		●			●	●	●		●
	HyperCat [20, 82]									●		●					●		●		●
	INTER-IoT [63, 65]	●	●		●	○				●		●	●				●		●		●
	IoT Hub [113]				●					●		●					●		●		●
	IoTivity [124]	●	●		●				●			●	●				●		●		●
	oneM2M [94, 160]				●					●		●	●			●	●		●		●
	symbIoTe [70, 89, 157]	●	●		●					●		●	●				●		●		●
	VSB [23, 24]			●		○				●		●					●		●		

Table 3.1.: Related Work Classification. None of the presented projects addresses all of the elaborated heterogeneities and, further, none of the approaches permits customisation of the alignment process (Reqs - requirements, Non-func. props - non-functional properties, Aw. - awareness, Tr. - translation, Di. - distribution, Gr. - granularity, Lo. - location, ● - fulfilled, ○ - partially fulfilled or mentioned without further specification).

Based on these results, the subsequent section demarcates the proposed framework in this thesis from existing approaches.

3.3. Placement of Thesis

From this insight, this thesis presents a customisable framework for achieving interoperability between pervasive platforms, granting developers a certain degree of freedom. The framework should be able to solve all of the heterogeneities while allowing to adjust the translation process, among other parts, for their needs. Therefore, the proposed framework should adhere to the presented requirements – extensibility, customisability, and dynamism – in order to support developers in their specific task. Further, many pervasive and IoT devices, using existing platforms and running applications and/or services, are already set up [65]. Existing and also future applications can greatly benefit from such an amount of services that is already there. In order to integrate those devices neither platforms [29] nor services/applications [92] must demand complex changes, and ideally no changes at all. Thus, the *transparent* awareness approach is employed. Regarding the translation model, the indirect one promises better extensibility compared to the direct model. Especially with various integrated platforms this makes sense and lowers the barrier for integrating new services. Therefore, the *indirect* translation model is used. Moreover, discovered services should be distributed to each integrated platform in order that every entity can benefit. Hence, the framework applies the *scattered* distribution model. Granularity basically defines if service matching happens syntactically or semantically. In order that no platform is disadvantaged, the proposed framework should give developers the opportunity to integrate either of the approaches, i.e., *coarse-grained* and *fine-grained*. This has to be incorporated in common abstractions. Furthermore, developers should have the choice of deploying an instance of the framework in the infrastructure, which makes sense in rather static domains like smart homes, and at-the-edge, which is reasonable in rather mobile environments. The placement of the proposed framework in this thesis is summarised in Figure 3.9.

In general, as long as there is no standard for interoperability (and also afterwards), developers should be able to use the middleware platform they prefer. Many proposed interoperability solutions include an API to create applications

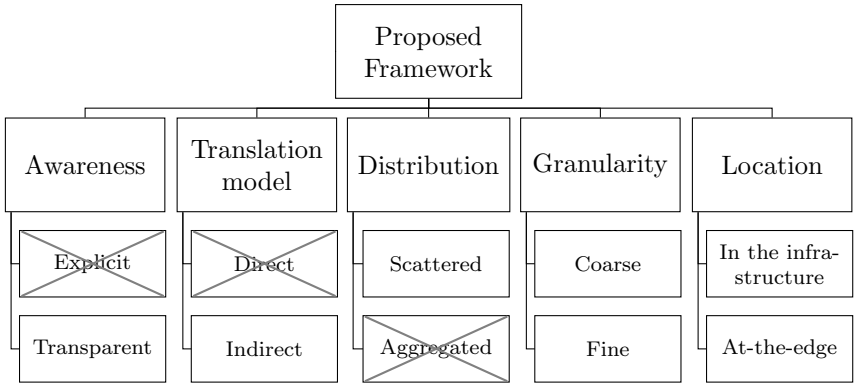


Figure 3.9.: Solution Classification in this Thesis. Whereas some dimensions are fixed, developers can choose for the granularity and location dimension.

and services. Thus, due to the fact that many of those solutions are proposed, the interoperability problem is only shifted to an upper level. The proposed framework in this thesis does not offer another middleware, but a solution where developers can easily integrate their services and continue their work as usual. If a standard should be established one day, it will be also possible to integrate it without making existing services and applications obsolete. Services can be specified in a common format, however, this is not the target. Therefore, there is no tool that supports this task. The framework's main objective is the alignment of protocols and messages between existing and upcoming platforms.

To summarise, this chapter introduced an evaluation framework that was used to assess existing interoperability frameworks and solutions. Further, a demarcation of this thesis from existing approaches was given. The next chapter presents the design of the proposed interoperability framework, including it system model.

4. An Interoperability Framework for Pervasive Computing Systems

The preceding chapter performed a thorough analysis of existing interoperability approaches and demarcated the approach in this thesis from the literature. Subsequently, this chapter presents the design of a general interoperability framework, called XWARE. First, Section 4.1 introduces the system model before Section 4.2 gives an overview on XWARE and its modules. Afterwards, Section 4.3 displays uniform abstractions that promote interoperability. Then, the different framework modules are presented with respect to their architecture and functioning. Besides, *XWARE* denotes the proposed framework, whereas an *XWARE instance* designates an interoperability instance that is instantiated and configured using XWARE. This chapter bases on and extends [147] and [149].

4.1. System Model

In this thesis, interoperability is provided within a federation. A *federation* consists of entities and interoperability instances. An *entity* is a piece of software or a device providing functionality and/or requesting functionality to/from other entities, i.e., an entity executes services and applications. Entities are running on a specific pervasive platform, such as BASE [12] or iPOJO [50]. A *pervasive platform* enables entities that use the same platform to discover each other and interact. The domain (e.g., vocabulary) within one platform is homogeneous in order that entities using the same platform understand each other. An *interoperability instance* enables discovery and interaction between entities running on different platforms, and is built with an interoperability framework. It has a set of platforms that it supports. For this, an *Interworking API* is required for each platform that should be supported. For the purpose of enabling discovery and interaction between entities, interoperability instances use knowledge for

transforming data. This knowledge is multi-faceted and includes service definitions, transformers, and domain knowledge. *Service definitions* define services in an intermediate language used by the interoperability instance. Each service that should be supported requires a service definition. *Transformers* translate service descriptions and messages between different platforms so that they understand each other. These transformers make use of the service definitions and domain knowledge of the different platforms. *Domain knowledge* contains, e.g., the used vocabulary within a platform. Furthermore, interoperability instances can have additional functionalities, e.g., context or conflict management. In the former case, a context model is created to have a unified context view which applications can use for their functioning. The latter is concerned with detecting and resolving conflicts between several pervasive applications, e.g., two pervasive applications that want to access the same monitor.

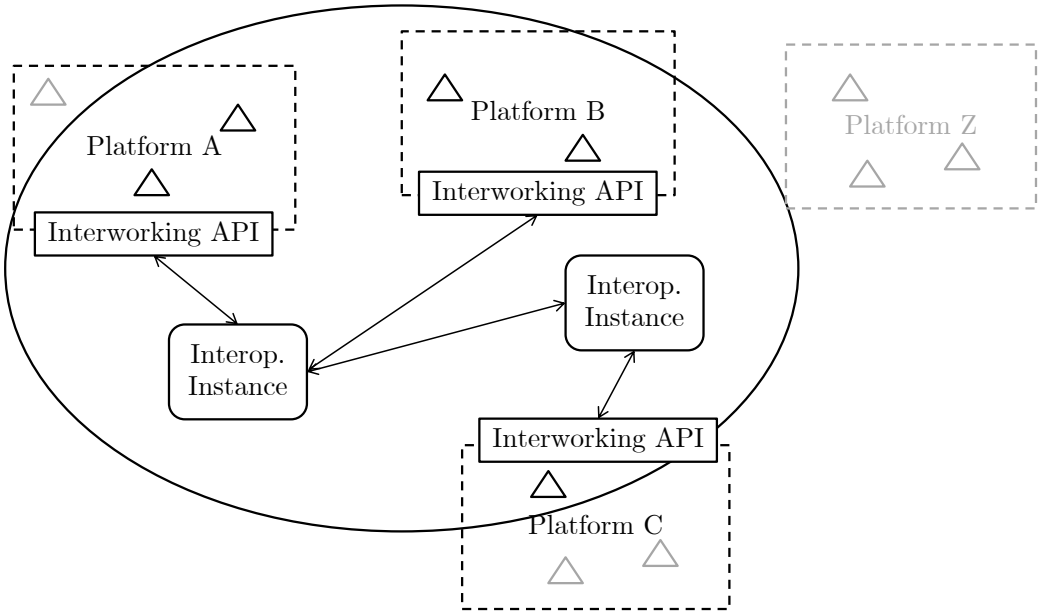


Figure 4.1.: System Model. A federation consists of entities and interoperability instances including service definitions, domain knowledge, transformers, and possibly additional functionalities. Interoperability instances enable communication between applications and defined services from supported platforms. Unsupported platforms (i.e., where no Interworking API exists, e.g., Platform Z) do not belong to the federation. Unsupported entities and platforms are depicted in grey.

Figure 4.1 summarises the notion of a federation. There, the left interoperability instance supports platforms A and B, whereas the right interoperability instance supports platform C. There is no Interworking API for platform Z, which means that it is not supported and, therefore, does not belong to the federation. Furthermore, the figure shows that although a platform is supported only specific services – those for which there exists a service definition – are included in the federation. In addition, there can be several interoperability instances in one federation. They can interact with each other, e.g., in order to advertise services or forward messages.

Because only services that are specified in the federation can be aligned by interoperability instances, a federation is uniquely defined by its interoperability instances including their supported platforms, service definitions, domain knowledge, and transformers, in addition to the present supported entities. Services for which no service definitions exist – maybe they should only be accessible by entities of the same platform – do not belong to the federation. However, they may have an influence on entities within the federation. A minimal federation consists of one interoperability instance consisting of its supported platforms, service definitions, domain knowledge, and transformers. Additional functionalities, as the ones mentioned, are optional extensions.

Within a federation, different responsibilities are taken over by interoperability instances (see Figure 4.2): protocol alignment, message alignment, service management, notification management, and potentially context management and conflict management. With *protocol alignment*, different protocols are made conform with each other in order that messages can be exchanged. If messages can be exchanged, they need to be understood by the communication partners to the end that the information can be used. This is the matter of *message alignment*. These two alignments – which basically are concerned with the introduced heterogeneities – allow to perform *service management*, i.e., service discovery and service access. Furthermore, they allow to do *notification management*, which means that entities can not only access service functionality, but also can get notified of updated data. As mentioned above, *context management* provides a unified context view, whereas *conflict management* detects and resolves conflicts within pervasive environments.

This thesis does not consider context and conflict management any further

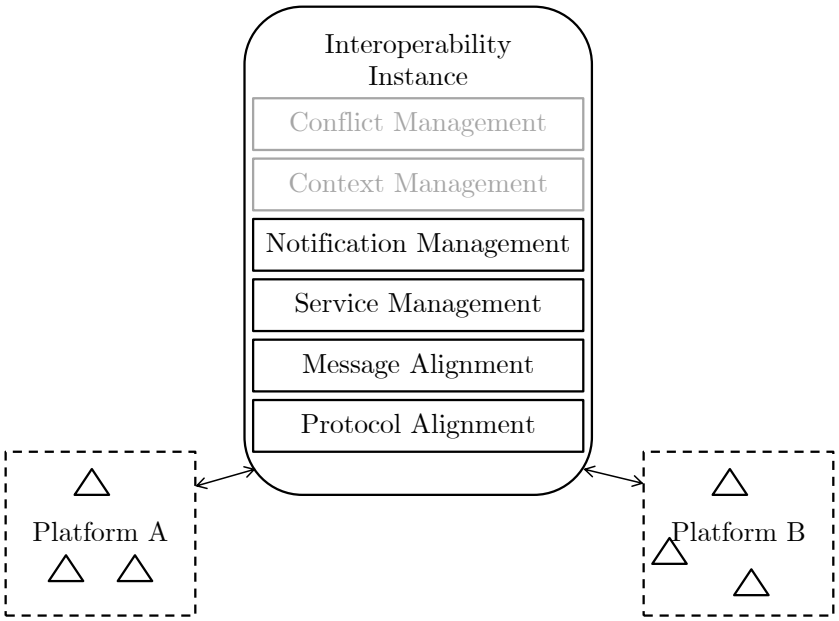


Figure 4.2.: Interoperability Instance Responsibilities. An interoperability instance is responsible for certain alignment and management tasks. This thesis does not consider context and conflict management.

(the interested reader is referred to [76], [107], or [148] for approaches regarding conflict management, and to [98] or [165] for approaches regarding context management). Nevertheless, it provides the basis for interoperability by taking protocol alignment, message alignment, service management, and notification management into account.

Having presented the system model, the next section introduces the XWARE interoperability framework.

4.2. Framework Overview

As seen above, achieving interoperability between pervasive platforms includes several steps. First, services need to be discoverable among different platforms. Second, entities have to be able to access these services. For both steps several alignments are necessary with respect to the heterogeneities presented in Section 3.1. However, an interoperability framework should not only deal with the align-

ment, but also adhere to the requirements. The following part briefly introduces the XWARE framework before going into detail on its components.

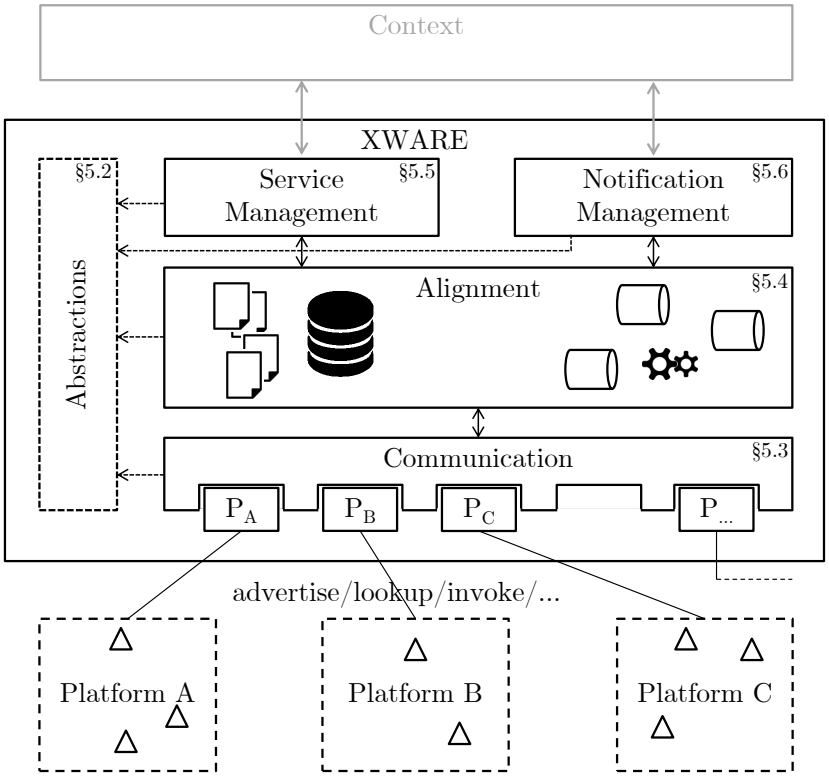


Figure 4.3.: Framework Overview. XWARE consists of four modules: communication, alignment, service management, and notification management. Each module covers one responsibility (P - plugin).

Figure 4.3 shows an overview of the XWARE framework including references to the sections where the corresponding part is detailed. The framework offers abstractions (see Section 4.3) for messages, services, service discovery, service access, and notification management. These abstractions allow for a uniform view and are required for supporting indirect transformation. Further, complying to the responsibilities of an interoperability instance (see Section 4.1), the functional components consist of communication, alignment, service management, and notification management modules. Each module covers one responsibility and can use the uniform abstractions. The *communication* module (see Section 4.4) performs protocol alignment, i.e., it mirrors a platform's protocol(s) and forwards important messages to the alignment module. The *alignment* module (see Section

4.5) is responsible for message alignment. Accordingly, it transforms messages between different platforms. The *service management* module (see Section 4.6) executes service management tasks, such as keeping track of the available services. The *notification management* module (Section 4.7) does, as the name suggests, notification management. Hence, it can notify entities of certain events. Further, modules may be added, e.g., for context management. Before presenting these modules in detail, the next part discusses the uniform abstractions.

4.3. Abstractions

This section presents uniform abstractions that facilitate translation of messages through a common view. This includes service, service discovery, service access, notification management, and message abstractions. Especially with respect to interaction heterogeneity and its differences in service discovery and service access, abstractions simplify transformations. Actually, most of the existing approaches only consider the CS interaction model (cf. Section 3.2 and [88]). Here, the three presented interaction paradigms – client-server, publish-subscribe, and tuple space (see Section 2.2) – are taken into account for developing the following abstractions, if appropriate.

4.3.1. Service Model

The literature review in Section 3.2 mentioned several service models. There, especially the PerSeSyn [116] and EASY [115] approaches examine a solution for semantic service discovery that could be also used with syntactic service discovery. Because developers should be free to use any type of service discovery they like to use, it makes sense to adopt one of these service models. The PerSeSyn service model includes the grounding of a service. A separation of the grounding from the service description makes sense as this information is platform-specific. Thus, the grounding should be stored separately. Furthermore, the PerSeSyn service model does not contain non-functional properties. On the other hand, the EASY service model provides support for non-functional properties, can be used for syntactic and semantic service discovery, and does not include grounding information. For completeness, further service models (e.g., from [69], [118],

[142], and [164]) include grounding information, or have been considered to be too domain-specific or inflexible. Consequently, they are not suitable. Hence, the EASY service model is adopted. However, a slight modification is introduced, as described in the following.

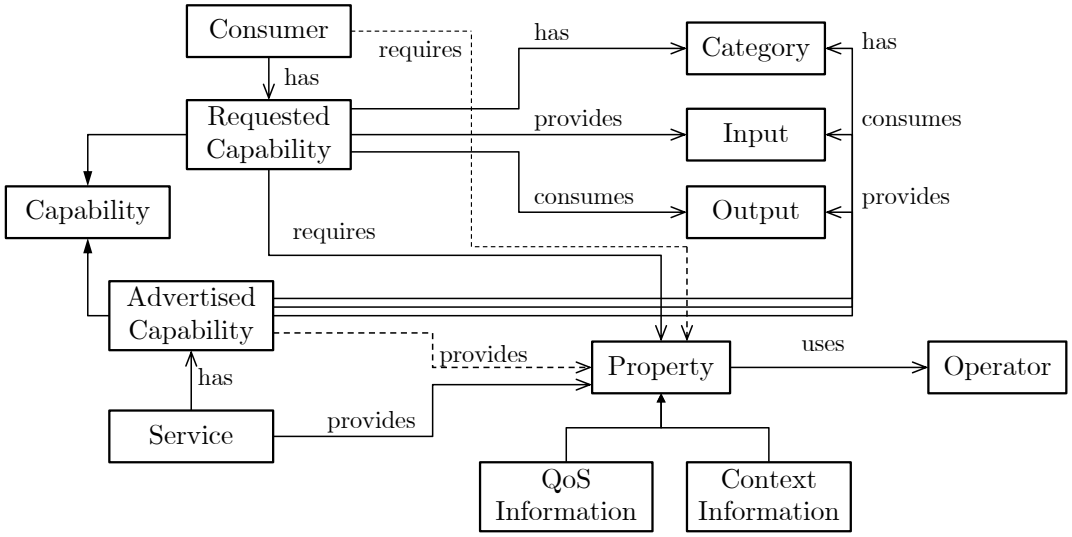


Figure 4.4.: Service Model. The service model is adopted from [115] and slightly modified. A service provides capabilities, whereas a consumer requests capabilities. A capability consists of a category, inputs, and outputs. Services and capabilities can have properties for specifying non-functional requirements. Modifications are shown by dashed arrows.

Figure 4.4 shows the modified service model. A *service* is characterised by its *advertised capabilities*, whereas a *consumer* is characterised by its *requested capabilities*. Capabilities have a *category*, an *input*, and an *output*. For instance, in a client-server system with syntactic discovery, a capability comprises the method name, the input (input parameters), and the output (result). The category specifies an ontology, if applicable. Services, consumers, and capabilities may provide/request non-functional *properties*, such as quality of service or context information. This could be, for example, information on the screen size or the location of a service. Properties use *operators* as comparison functions. Capabilities are the functional specification of a service, whereas properties are the non-functional specification. In the original service model of EASY, only capabilities include non-functional properties. However, a service/consumer can also provide/require such properties (cf. [119] or [164]).

XWARE uses the introduced service model as intermediate service representation. After having introduced the service abstraction, the next section presents the service discovery model.

4.3.2. Service Discovery Model

Service discovery aims at finding services for potential interaction. As seen in Section 2.2, different interaction models additionally deviate in their service discovery patterns. To the best of the author's knowledge, existing approaches for service discovery interoperability only consider service discovery of platforms using a client-server (CS) interaction, e.g., [17], [29], [45], [57], [71], [106], [115], [142], or [162]. This is presumably due to the fact that most of the approaches do not consider interaction paradigm heterogeneity. For both publish-subscribe (PS) as well as tuple space (TS) interaction service discovery is handled more loosely due to their characteristics of loose space and time coupling. Since platforms using CS interaction require to know their communication partner, it is crucial that communication partners and their capabilities are made available. In the following, a uniform pattern and model for service discovery among interaction paradigms are proposed.

Section 2.2 introduced the different interaction paradigms and their service discovery approaches. Their service discovery patterns are different, but also show some similarities, i.e., each interaction model uses advertisement messages, and CS- and TS-based systems may use a lookup mechanism. However, regarding advertisements, the content of such a message differs for the interaction models, i.e., event categories are advertised in the PS model, whereas service descriptions are advertised in the CS and TS models. To encounter this problem, event categories can be mapped to a service. This is feasible because, according to the service model, a service contains one or more service capabilities. A capability in a PS-based system can actually be represented by the event category (see Section 4.3.3). Thus, an inference from an event capability to a service is possible. For instance, a PS-based temperature provider periodically publishes the sensed temperature using the event category *Temperature*. From that, one can conclude that the service is a temperature sensor. Even if there are no explicit advertisements, the same mapping is possible with the actual events. Considering

the TS interaction paradigm, advertisements can either contain provided services or not. Whereas the former case is simple, the latter case requires changes in the pervasive platform in order to make services available to other platforms. In general, if a platform does not advertise its services, entities of other platforms are not able to consume them. However, entities of this platform can still access services of other platforms. Thus, here it is assumed that all entities that want to actively participate in the federation use a proper service discovery mechanism. By 'proper' it is meant that services are discoverable.

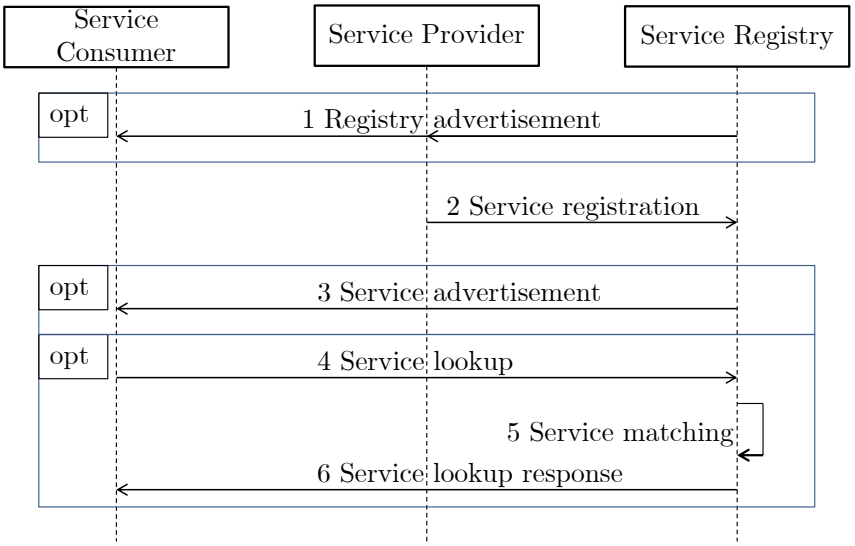


Figure 4.5.: Service Discovery Pattern. Service registries may advertise themselves to other entities. Service providers register their services at the service registry. Services are either advertised by the service registry or specific services are sent in response to a service request (opt - optional).

From the differences and similarities in the service discovery mechanisms, a general pattern can be inferred based on a SOA. Furthermore, the approach for a generic service discovery design for CS-based platforms in [59] serves as basis. There, the authors identify two patterns depending on the registry architecture of platforms: centralised or distributed. In the centralised approach, there is one central service registry, whereas in the distributed approach, each entity runs its own service registry. In the former case, an additional registry advertisement message is sent from the service registry to other entities. Also, the communication form may differ message-wise between the two designs, i.e., unicast or multicast.

Here, these two patterns are aggregated into one and incorporate service discovery for CS-, PS-, and TS-based platforms (see Figure 4.5). In PS- and TS-based platforms, the broker and, respectively the TS, can be seen as service registry because they receive advertisements for services. According to [59], depending on the registry organisation, the directory advertises itself so that entities get to know it and its location (Figure 4.5, step 1). Then, service providers can register their services (step 2). In PS- and TS-based discovery, services are not advertised to consumers at all. This happens only in CS-based systems (step 3). However, TS-based consumers can look service information up at the TS using tuple templates. This is similar to a service lookup in CS-based systems. Then, the service registry performs a service matching and sends back the reply (steps 4, 5, and 6).

Furthermore, two service discovery models are presented in [92]: service lookup translation and service registration translation. Whereas only lookup requests are translated and passed between domains in the former one, only registration messages are translated and forwarded between domains in the latter one. Considering the fact that registration messages are rather static, the latter approach seems better suited for pervasive environments, reducing translation overhead. An advantage of registration translation is also that platforms can continue using their own service matching algorithms.

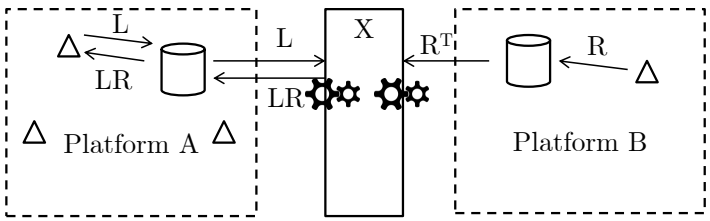


Figure 4.6.: Service Discovery Model. Only service registrations are translated in the interoperability instance (X). Service queries are processed without translation. The interoperability instance may use different service matching algorithms per platform. All translations happen inside of the interoperability instance (R - registration, L - lookup, LR - lookup response, a superscript T indicates that a translation takes place in this step).

Here, in order to comply to the service discovery pattern, registration messages are translated, but not automatically forwarded. The platform-specific patterns are then applied, i.e., registrations may be advertised or services may be looked

up. Lookup requests are not translated, but processed with the translated services. Hence, each platform can use its specific matching algorithm which can be based on syntactic or semantic matching due to the employed service model. Consequently, an XWARE instance may serve as 'global' service registry containing services from each supported platform in the federation. Figure 4.6 depicts this approach.

This section presented a pattern and model for service discovery. The subsequent section introduces the service access model.

4.3.3. Service Access Model

After services have been discovered, they possibly want to be accessed by other entities. Service access happens through application and application response messages. Depending on the interaction model, the content of these messages differ (see Section 2.2). The differences and a way to cope with this are explained in the following.

Interaction Paradigm	Action	Input	Output
CS	Method name	Arguments	Return value
PS	Event Category	Event	Event
TS	Template	Tuple	Tuple

Table 4.1.: Message Content Abstraction. Action, input, and output are abstract constructs that allow to translate application message content between interaction paradigms.

Although the content of application and application response messages contain very diverse information among interaction paradigms, the information concentrates basically on one or several of the following aspects [14]: action, input, and output. Whereas the *action* provides information on a method or provided/required data, the *input* and *output* add additional information for processing the action. Table 4.1 shows how these aspects can be mapped to the concrete interaction paradigm messages. In CS interaction, a message that contains a method name (action) and arguments (input) is usually sent. The communication partner, depending if a result is expected, answers with a return value (output). In PS interaction, a consumer subscribes at a broker by sending an event category

(action). The provider publishes events (input) to the broker including the event category (action). The broker notifies matching consumers of the event (output). In TS interaction, providers write tuples (input) into the tuple space. Consumers can take/read tuples (output) from the TS by using templates (action). By adopting this abstraction, translation of application message content is possible, despite different interaction paradigms.

Interaction Paradigm	**One-way**	**Two-way**
Abstract	post	post-get
CS	one-way message	request-response
PS	publish	subscribe
TS	out	in/take

Table 4.2.: Interaction Semantics Abstraction. The post construct does not expect a response, whereas the post-get construct does.

Furthermore, the different operations have different interaction semantics that also need to be taken into account. Interaction semantics may contain a different amount of information depending on the paradigm [68]. Across the paradigms, there are operations where one or several response messages are expected – CS: request-response, PS: subscribe, TS: in/take – and there are operations where no response is expected – CS: one-way message, PS: publish, TS: out. Thus, they can be abstracted to an operation awaiting responses (post-get) and an operation without responses (post), in accordance with [23]. Semantics may further include a lease time to indicate after what time a subscribe (PS) or in/take (TS) message should be aborted. Table 4.2 summarises the abstract operation and their specific mappings.

Another difference between service access of the distinct interaction models is that communication partners are different for the operations. Consumer and provider interact directly in CS interaction, independent of the operation. In PS or TS interaction, consumers and providers communicate indirectly via the broker or TS, respectively. Thus, the source and target of these messages differ. Table 4.3 gives an overview on how to map the source and target between the different interaction paradigms.

Based on these service access abstractions, an application message consists of the operation including action, input, and output, interaction semantics, and

Interaction Paradigm	Interaction Semantics	Source	Target
CS	request-response	Consumer	Provider
	one-way message	Consumer	Provider
PS	subscribe	Consumer	Broker
	publish	Provider	Broker
TS	write	Provider	Tuple space
	in/read	Tuple space	Consumer

Table 4.3.: Communication Partner Abstraction. Source and target of a message differ depending on the interaction paradigm and semantics.

communication partners. The next section presents a uniform notification management model.

4.3.4. Notification Management Model

The elaborated disparities in notification systems (see Section 3.1.1) demand means in order to overcome these differences. Based on those differences and, further, on the general characteristics of notification systems (see Section 2.3), Figure 4.7 depicts a general notification management model. In addition to a notification management, several consumers (C) and providers (P) of different nature (push, pull, and non-supporting) and pervasive platforms are visible. It is assumed that the notification management knows about a platform's support for notifications, its delivery modes, and notification scheme, in order to do alignment of messages. Supporting platforms use a local notification system. Local notification systems have to communicate with the notification management without requiring changes. There are two possibilities to achieve this: message interception and proxy. The first option is that the management intercepts messages at the notification system. The other option is that the management acts transparently as proxy, i.e., as notification provider and consumer, subscribes to all events at a local notification system, and also forwards every upcoming event. In the following, message interception is assumed (such messages are indicated with brackets). However, the second option is also feasible for this model, possibly including more messages due to different delivery modes of local notification systems. Consumers that are interested in certain events (E) have to indicate their interest by sending a subscription (S) to their local notification system, and

hence, to the notification management. Basically, there are two choices how to handle incoming subscriptions at the notification management: 1) translate and forward the subscription or 2) store the subscription and subscribe at local notification systems. With the former option, event matching happens at the local notification systems, and only events for which the notification management has subscribed are forwarded. In the latter approach, event matching takes place at the notification management. Thus, each event is known at the notification management, which is the reason for employing this option. Besides, consumers of non-supporting platforms cannot participate without adjusting them.

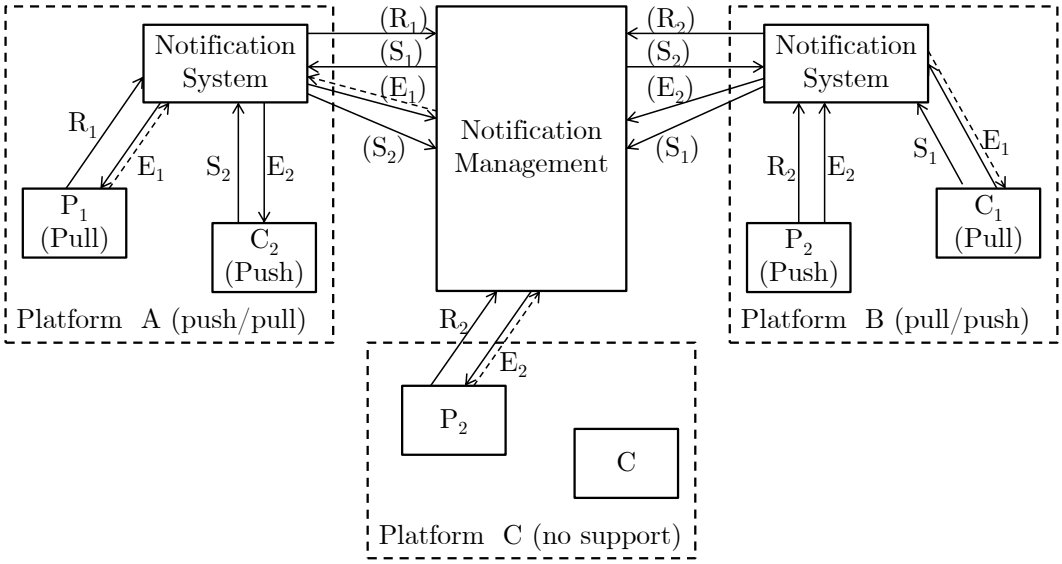


Figure 4.7.: Notification Management Model. The notification management enables inter-platform notifications. It deals with platforms that differ in there delivery mode, event category types, and notification support. Communication between local notification systems and the notification management happens either transparently or explicitly (C - consumer, P - provider, R - registration, S - subscription, E - event, distinct subscript numbers indicate different event categories).

Furthermore, in order to be aware of the providers, they need to register at the notification management component. This happens automatically through the registration messages (R) from service discovery. Thus, no overhead or changes are implied at those entities. If the provider supports notifications, events are automatically received by the notification management through message interception. In case of using the proxy approach, the notification management com-

ponent subscribes at the local notification management for all events of that provider. Then, events are automatically received from providers using the push mechanism, or they need to be pulled periodically from providers using the pull mechanism. If the provider does not support notifications, a periodic polling is initiated for new values, i.e., events. Incoming events are then matched against the active subscriptions and forwarded to the respective local notification systems where the event is distributed to the actual consumers. In the figure, the subscript numbers indicate different event categories.

So far, the two issues of aligning messages, especially event category types, e.g., channels to subjects, and polling at non-supporting platforms are implicitly assumed here. Their working is explained in detail in Sections 4.5 and 4.7, respectively. The next section introduces a uniform message abstraction for the messages known from the service discovery, service access, and notification management models. Next, a uniform abstraction for messages is introduced.

4.3.5. Message Abstraction

The former sections showed that several types of messages can occur in pervasive environments. For service discovery, there are advertisement, registration (and deregistration), as well as lookup and lookup response messages. For service access, application and application response messages exist. Furthermore, for notification management, there are subscription and unsubscription, and event messages. Apart from the different purposes of these messages, they also contain similar information, as shown in the following.

For the design of a query language, Finin *et al.* use a protocol approach [56]. They propose a protocol stack with three layers: declaration, content, and communication. Accordingly, a message is divided into those three parts. Here, message abstraction takes over this logical separation from [56] and extends it with an interaction layer for incorporating differences in interaction semantics, as follows. The *declaration* provides basic message information, such as the message type and identifier. The *content* represents the actual content of a message. The *communication* layer holds the two communication partners for directed messages, i.e., source and target. The *interaction* layer is only required for application events and stores information on the interaction model, e.g., non-blocking

client-server interaction. Table 4.4 gives an overview on the layers including an example.

Layer	Message	Information	Example
Declaration	All messages	Type, identifier	Application, 23
Content	All messages	Actual content (depends on message type)	void setLight(true)
Communication	All messages	Source, target	LightConsumer, LightProvider
Interaction	Application (response)	Interaction model, interaction semantics	CS, one-way

Table 4.4.: Message Abstraction. All messages include declaration, content, and communication information, whereas only application (response) messages include interaction information.

This section elaborated several uniform abstractions that support the integration of different platforms. Therefore, the abstractions are used in the different modules of the framework which are discussed in the following section, starting with the communication module.

4.4. Communication

The general responsibilities of the communication module are the interception of messages, the actual interaction with entities, as well as performing service discovery. In order that support for platforms can be easily added or removed at design time, without having any effect on other platforms, the communication module uses a plugin-based approach. In general, a plugin is responsible for supporting one platform, and hence, represents the platform's Interworking API. Among other tasks, plugins mimic platform-specific service discovery and access mechanisms. For this reason, their internal architecture is derived from the service discovery and access models (see Section 4.3), and is presented in Figure 4.8.

The *connection manager* holds the actual connections to services and applications of a specific platform. Hence, it is responsible for sending and receiving

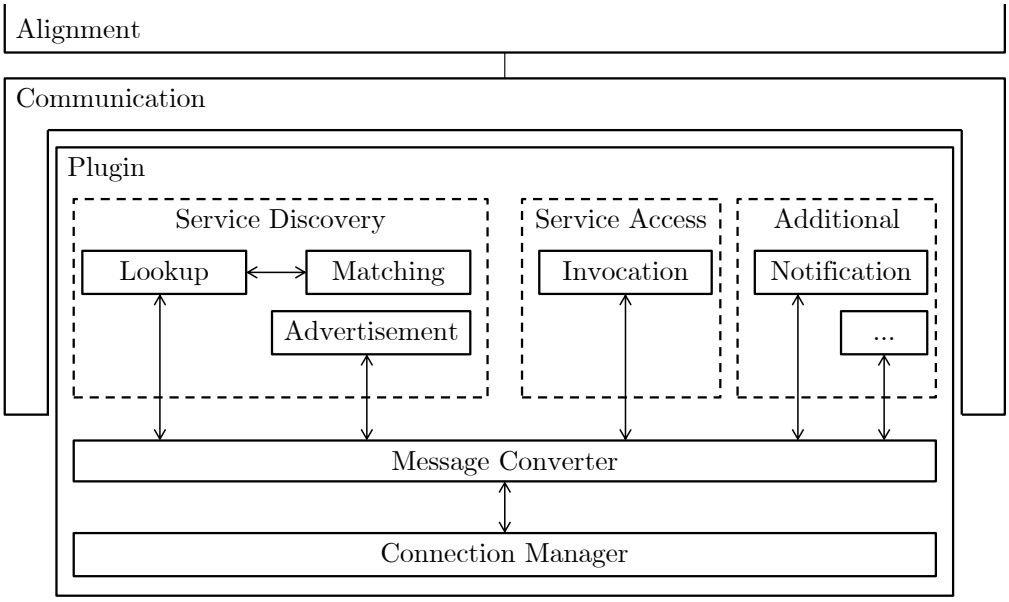


Figure 4.8.: Plugin Architecture. Plugins are integrated into the communication module. The connection manager communicates with entities and uses the message converter for conversion between the intermediate message abstraction and platform-specific message formats.

messages to/from entities. It also intercepts service discovery messages, i.e., device or service advertisement messages as these are usually sent via multicast (cf. [29]). Each platform uses a unique pair of multicast address and port to multicast advertisements [29] in order that entities become aware of each other's presence. Accordingly, the connection manager joins a respective multicast group to intercept these messages. Before outgoing messages are sent to entities or incoming messages are processed, they need to be converted. For this, the connection manager works with the message converter. The *message converter* converts the representation of incoming messages to the abstract one, and the representation of outgoing messages to the platform-specific one. However, no content translation takes place so far. Each plugin requires its own platform-specific message converter. A message converter's implementation has to implement the following interface which is derived from its conversion directions:

```
interface IMessageConverter  {
    Object processAbstractMessage(Message msg);
    Message processSpecificMessage(Object data);
}.
```

After this conversion step, an outgoing message is handed to the connection manager for sending, whereas an incoming message, depending on its type, is distributed to the appropriate component, e.g., a lookup request is forwarded to the lookup component. Based on the service discovery pattern (see Section 4.3.2), the three components for service discovery become clear. The *advertisement* component advertises available services and possibly the registry. It further listens to incoming registration messages and passes them to the alignment module for translation of the contained services. Additionally, it may react to incoming registry advertisements by sending lookup requests to the corresponding registry. The *lookup* component waits for incoming lookup requests and replies with available services that match the request. The *matching* component performs the matching process. According to the service discovery model, lookups are not translated and forwarded to other platforms, but handled in the XWARE instance, i.e., in a plugin. Thus, the implementation of the matching component can vary between plugins. All matching components require the implementation of the following interface:

```
interface IMatching  {
    ServiceDescrs lookup(MatchRequest request);
}.
```

The *invocation* component stores information on application messages in order to relate requests and responses. Additional components, such as a *notification* component, process component-specific messages and store related information. Further components may be added if necessary, e.g., for routing of messages in an *ad hoc*-based setting.

Regarding the heterogeneities between platforms, communication heterogeneity (H1) can be addressed by implementing several plugins for the same platform but different technologies. Plugins, together with the proposed service discovery pattern and model (see Section 4.3.2), permit to overcome service discovery heterogeneity (H2) apart from the semantics of service descriptions, as no content transformation takes place here. Interaction model instantiation heterogeneity, which is part of interaction heterogeneity (H3), is addressed by mimicking the platform interactions and extracting information. Moreover, the message converter enables mastering syntactic data heterogeneity (H4).

Messages that possibly require alignment, e.g., in order to forward them to another platform or module, are handed over to the alignment module. Such messages include registration (and deregistration), application and application response, subscription (and unsubscription), and event messages. The next section explains the alignment module and process in detail.

4.5. Alignment

Alignment can be defined as 'a state of agreement or cooperation among persons, groups, nations, etc., with a common cause or viewpoint' [44]. This definition fits quite well by including also pervasive middleware platforms as agreement partners. These platforms usually have a common viewpoint, but are not able to communicate due to the presented heterogeneities (see Section 3.1). Aligning these heterogeneities, i.e., by using common abstractions (see Section 4.3), makes interaction possible. The common abstractions represent the agreement between platforms. Figure 4.9 depicts the general architecture of the alignment module.

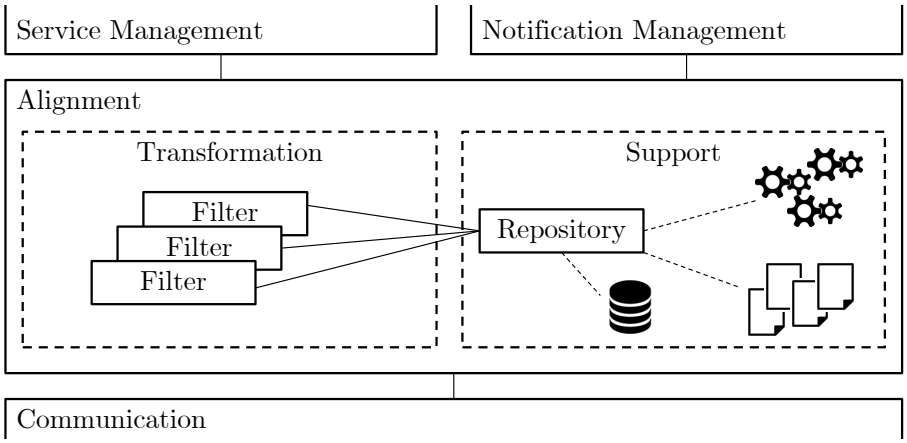


Figure 4.9.: Alignment Architecture. Filters perform independent transformation steps. Therefore, they may access the repository which stores service definitions, domain knowledge, and transformers.

The alignment module gets input from the communication, service management, or notification management modules. Each arriving message, independent of the source module, requires transformation in order to forward it to a platform (via the communication module), or an upper-level module. Although such a message is already in the abstract message format, there still are several

heterogeneities remaining. Therefore, transformations are necessary. For this, the alignment module provides the following interface which offers a method for performing message type-dependent translations and pass aligned messages to appropriate modules:

```
interface IAlignment  {
    void process(Message msg);
}.
```

Transformations are performed with so-called filters and a repository, allowing for customisation of the alignment process. Filters execute independent transformation tasks with the help of the repository. The next section explains the transformation process in detail before presenting means for the purpose of managing further heterogeneities.

4.5.1. Transformation Model

In order to provide customisation of the transformation process, employing the pipes and filters pattern [31] seems to be appropriate. The *pipes and filters* pattern defines an architectural style for processing a data stream by dividing large tasks into a chain of small, independent processing steps [31]. A *filter* represents a processing step and has an input as well as an output. Input and output types are the same for each filter. During a processing step – or an application of the filter to the data – incoming data is converted. Conversion can be through extraction, addition, or replacement of data, and moreover, is defined by the specific filter. A *pipe* connects two filters – or processing steps. A *source* is a component that can write input data to the initial pipe, whereas a *sink* receives output data from the last pipe. Figure 4.10 illustrates the architectural structure of the pipes and filters pattern. Here, filters perform independent tasks as well. Though, during the conversion process filters can make direct use of the data. For instance, a filter can convert a service description, and can directly hand it over to the service management module.

The pipes and filters pattern allows for extensibility and customisability as described in the following. New filters can be easily added or existing ones replaced because the input and output types are always the same. Here, input as well as output refers to a set of messages. Thus, if the transformation process requires

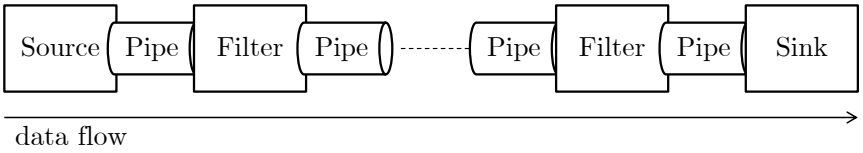


Figure 4.10.: Pipes and Filters Pattern (cf. [31]). Filters represent small, independent processing steps while pipes connect them.

changes, it is simple to change the process or only part of it. Further, through rearrangement of filters, different data streams – or messages – can be processed according to their specific characteristics. This way, the pipes and filters pattern can process data in a context-aware manner, where the context consists of the data type. Recombination of filters allows to re-use existing filters in similar systems (i.e., a federation where a different transformation process is required).

Here, filters provide the logical flow for different transformation processes. For this purpose, some filters make use of the repository which stores service definitions, domain knowledge, and transformers to support the process. Transformers execute rather simple mapping functions, possibly using the service definitions and domain knowledge. Therefore, a transformer requires the implementation of the following interface, for transforming service descriptions, operations, and non-functional properties:

```
interface ITransformer  {
    ServiceDescr transformServiceDescr(
        String from, String to, ServiceDescr sd);
    Operations transformOperation(
        String from, String to, Operation op);
    Object transformEvent(
        String from, String to, Object event)
    Properties transformProperties(
        String from, String to, Properties props);
}.
```

Existing interoperability frameworks do not offer any choice to the developer with respect to the transformation mechanism. Developers are bound to the provided tools or mechanisms (cf. Section 3.2) that are usually tailored to specific characteristics. If a service shows divergent characteristics, the built-in transformation mechanism cannot be used. This is why this approach offers, apart

from an integrated automatic alignment process, the possibility to specify transformers manually. Developers then have to specify a transformer per service and platform. Both kind of transformers, automatic and manual, are based on the `ITransformer` interface. Furthermore, the integrated automatic alignment process can be modified, replaced, and extended – in parts or completely.

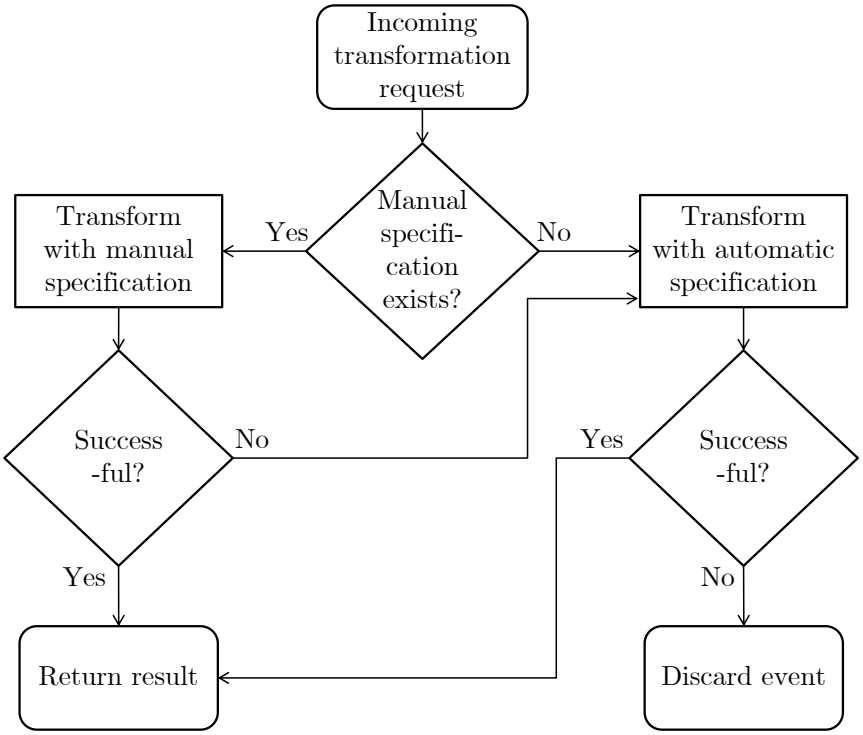


Figure 4.11.: Transformation Selection. The manual transformation has a higher priority as the automatic one. Thus, if such a specification exists it is employed. The automatic transformation remains as backup strategy.

As there are two possible transformation algorithms (automatic or manual), a transformation selector is employed. The manual transformation specification is prioritised over the automatic one. Consequently, the transformation selector first checks if there exists a manual specification. If so, it uses it. Otherwise, the automatic algorithm performs the transformation. Also, if the manual transformation fails, the automatic one is tried. If the automatic transformation fails, no further backup strategy exists. The filter then has to decide if the message is handed over unprocessed to the next filter, or if it is discarded. Since the probability that a message is understood by another platform, although some

transformation steps are missing, is very low, the message is discarded by the filter. Figure 4.11 illustrates this procedure.

The alignment module decides which filters are applied on a message based on the message type. A registration message varies in the service description semantics (H2), content (H4), and non-functional properties (H6). An application message differs in the interaction models and semantics (H3), content (H4), operations and their granularity (H5), and non-functional properties (H6). Subscription and event messages require alignment for differences in their content (H4) as well as event categories and schemes (H7). Thus, they share semantic data heterogeneity with respect to the message content. Because solving this issue is dependent on the message, there is no separate filter proposed for this, but it should be done by each filter if required. The same is actually true for non-functional properties. Furthermore, in accordance with the service access model (see Section 4.3.3), a filter for the transformation of service identifiers, i.e., communication partners, is introduced. Consequently, the following filters are derived and adhere to the heterogeneities: service identifier, discovery, interaction, application, and notification. Table 4.5 summarises the message types, their corresponding filters, and the addressed heterogeneities.

Message Type	**Filters**	**Heterogeneities**
Registration (and deregistration)	Discovery	H2, H4, H6
Application (and application response)	Service ID Interaction Application	H3, H4, H5, H6
Subscription	Service ID Notification	H4, H7
Event	Service ID Notification	H4, H7

Table 4.5.: Message Type to Filters Mapping. Depending on the message type, the alignment module employs different filters for transformation (ID - identifier).

Figure 4.12 shows the exemplary procedure for aligning an application message using an indirect translation. There, a CS consumer sends a request for the light state to a PS provider. In a first step, the service identifiers (source and target) are translated into the intermediate UUID representation by the service

identifier filter. Then, the interaction filter transforms the interaction model into the abstract semantics, i.e., from request-response to post-get. Subsequently, the application filter transforms the message content. The message is now completely represented by the intermediate representation and semantics. The same steps are then applied in reverse order to transform the message into the target semantics.

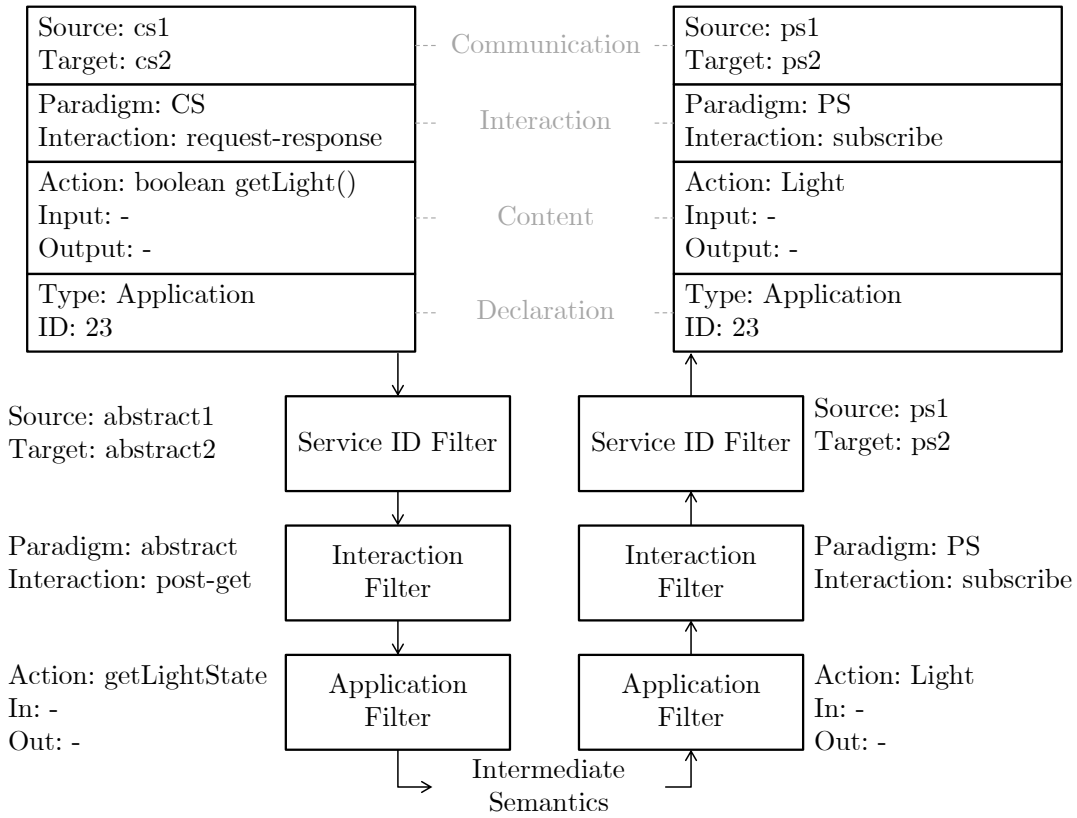


Figure 4.12.: Exemplary Transformation Process of an Application Message. Filters process messages step by step, possibly applying several filters. Here, the service identifier (ID) filter transforms the endpoints, before the interaction filter aligns the interaction semantics. The application filter translates the actual content. The message is then completely in the intermediate semantics. Subsequently, the same filters are applied in reverse order to transform the message into the target semantics.

Before having a closer look at the different filters, the service definitions are presented due to the fact that they are used as basis for the automatic transformation algorithms.

4.5.2. Service Definition

Service definitions define which services are supported in a federation. Therefore, a service definition needs to exist for each supported platform and for the intermediate representation, in case of indirect translation. They are stored in the repository. Here, two possibilities are offered on how these service definitions are specified: by files based on the Web Service Description Language (WSDL) [40] or by code. The former solution goes hand in hand with the automatic transformation tool, whereas the latter one should be used if manual transformation is desired/required, i.e., if the automatic tool is not able to do the transformations. When using manual transformation, the service definition is implicitly contained in the transformer specification.

WSDL is a language based on XML that can be used to describe web services. It is widely known as well as used (e.g., [57] or [71]) and provides a fairly human-readable notation [42]. A WSDL description file contains an abstract and a concrete part. Whereas the abstract part describes the functional properties of a service, the concrete part determines protocol and communication details. The functional part includes sections for data types, messages (capabilities), and port types (operations). A `message` element has a `name` attribute and contains `part` elements. A `part` element represents an argument and has a `name` and a `type` attribute. A `portType` element is defined by its `name` attribute. It contains one or several `operation` elements. An `operation` element also has a `name` attribute and an `input` as well as an `output` element. These elements have a `message` attribute. Thus, the port types represent the way of interaction. Because the concrete part is not relevant here (cf. Section 4.3.1), it is not further explained.

For the automatic transformation tool, WSDL files serve as basis. However, they are extended with further attributes, e.g., the `maps` attribute which is used to specify a mapping between platform-specific and abstract service definitions. Intermediate definitions do not require any `maps` attributes. While reading out the files at start-up, mappings are established between platform-specific and the intermediate semantics. Placing the mapping specification in the intermediate definition is not feasible because it would require a mapping for each supported platform, and thus, reduces transformer extensibility. Those extensions are introduced and explained successively when required in the upcoming sections.

Henceforth, extended WSDL descriptions are called *XWSDL definitions*.

Figure 4.13 shows an extract of a simple light service of an XWSDL definition. It has two `message` elements: one for getting the state of the light (on/off), the other is the response to the first message. This is also indicated by the `port type` element since it has an `input` as well as an `output` element.

```
<definitions name="SimpleLight">
  <message name="getState" />
  <message name="getStateResponse" >
    <part name="result" type="Boolean" />
  </message>
  ...
  <portType
      name="SimpleLightPortType">
    <operation name="getStateOperation" >
      <input message="getState" />
      <output message=getStateResponse" />
    </operation>
    ...
  </portType>
</definitions>
```

Figure 4.13.: XWSDL Example: Extract of an Intermediate Light Service. WSDL serves as basis. The service provides an operation for requesting the light state.

In conclusion, two ways of specifying service definitions exist, and the developer is responsible for doing this. Further, each service requires a service definition for the source platform, the intermediate representation, and the target platform. Knowing the basic transformation concepts, the next sections present several transformation steps, starting with service description transformation.

4.5.3. Service Description Transformation

According to the service discovery model (see Section 4.3.2), XWARE instances advertise service descriptions to platforms. Therefore, they must be in the platform-specific language, and thus, they require transformation. For this, the use of the automatic tool or manual transformation is possible.

For the automatic transformation, the `definitions` element of an XWSDL definition is extended with a `maps` attribute. Figure 4.14 shows an extract of a BASE-specific description. The introduced `maps` attribute is underlined. The attribute indicates that this service definition can be transformed into the intermediate `SimpleLight` definition (see Figure 4.13). Values of the `maps` attributes must match exactly the respective values of the intermediate definition. The automatic tool performs this mapping and can then read out the matching service description, if present.

```
<definitions name="base.light.ILight"
          maps="SimpleLight" >
  <message name="boolean getState()" >
  <message name="getResponse" >
    <part name="result" type="Boolean" />
  </message>
  ...
  <portType
      name="SimpleLightPortType" >
    <operation name="getOperation" >
      <input message="boolean getState()" />
      <output message=getResponse" />
    </operation>
    ...
  </portType>
</definitions>
```

Figure 4.14.: XWSDL Example: Extract of a BASE-specific Light Service. For transforming service descriptions, the `maps` attribute (underlined) extends the basic XWSDL definitions.

For the manual transformation, developers have to implement the method `ServiceDescr transformServiceDescr(...)`. The transformer is then automatically used by the transformation selection.

The discovery filter is responsible for this transformation. It is used when a service registration event is received in the alignment module in order to transform the original service description. Applying this filter solves the remaining part of service discovery heterogeneity (H2), i.e., distinct service description semantics. The next section goes into detail about service identifier transformation.

4.5.4. Service Identifier Transformation

Service identifiers refer to the source and target of messages. They are usually represented by some kind of identifiers. For example, in the BASE [12] platform, an identifier may be *756dc02333b09d220000000000000000000000000 ffffffffffffff*, whereas in Limone [60], it may be *MyDevice-192-168-1-123-4000:a0:LightService*. Since these endpoints are included in service advertisement and service access messages, their transformation is mandatory. To the best of the author's knowledge, most other approaches neglect this transformation (or implicitly assume it) although it is essential for interoperable service discovery and access.

Here, the foundation is a mapping between endpoint representations using an intermediate representation. The intermediate representation must be unique within a federation. Therefore, universally unique identifiers [102] (UUIDs) seem to be a reasonable choice, and are also suggested in [106]. The mapping itself is performed with the help of the service management module which is explained in Section 4.6. When a service is registered, the endpoint mapping is created. Due to the possibility of communication between interoperability instances, it can happen that the same service is discovered by several interoperability instances. For a unique identification among them, name-based UUIDs [102] are used that depend on the original registry and service identifier. Consequently, each interoperability generates the same UUID for such a service.

Service identifier transformation is performed by the service identifier filter. It covers part of semantic data heterogeneity (H4). The next transformation step addresses interaction heterogeneity.

4.5.5. Interaction Transformation

Interaction heterogeneity deals with diverse interaction models. Most existing approaches do only consider the client-server model and neglect interaction heterogeneity completely [88]. In [14], [23], and [68], this heterogeneity is addressed by using a formal description. Here, partially based on these works, a graphical combination is performed. Considering not only the interaction models but also their operations, the following overview is more complete than the mentioned approaches. From the service access model (see Section 4.3.3), it is known

that application message content can be abstracted as <action, input, output>. Further, the interaction semantics can be mapped using the intermediate semantics post and post-get. However, it is not as simple as that, because operations cannot be translated one to one. Therefore, the basic interaction patterns are revisited separately by consumer- and provider-side interaction in the following. Subsequently, they are visually combined.

Figure 4.15 shows the basic interaction patterns. It is easy to notice that the interaction patterns are mirrored on consumer- and provider-side. However, it serves as better illustration because these building bricks are aggregated one by one together with an intermediate interoperability instance.

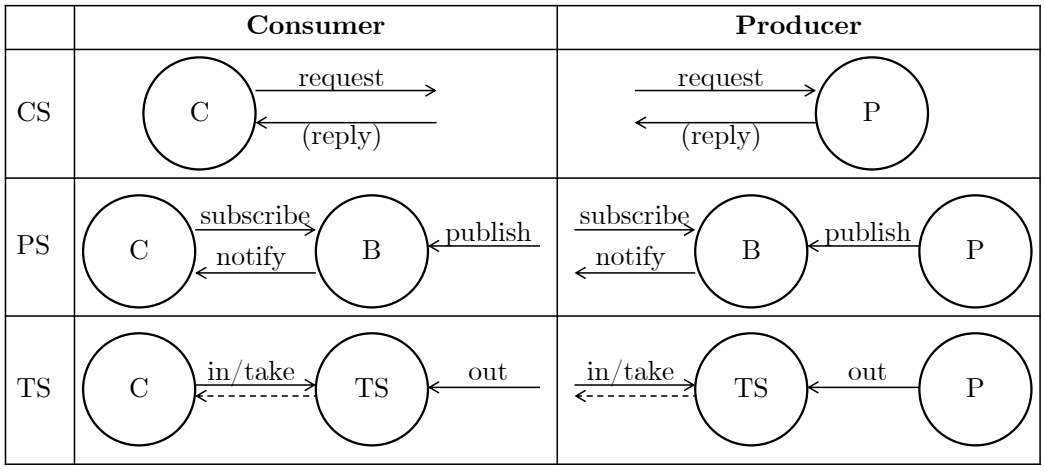
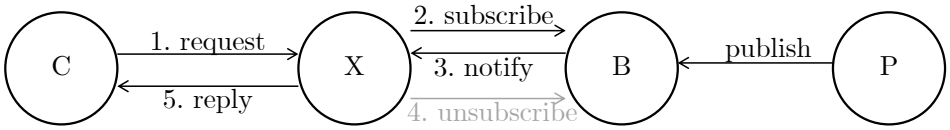


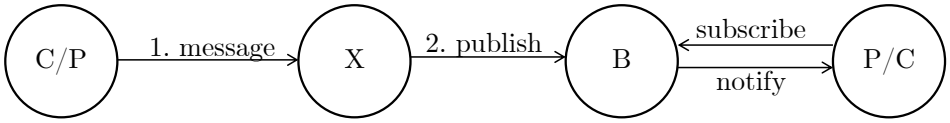
Figure 4.15.: Revisit: Common Interactions. This figure summarises the common interactions of different interaction paradigms (C - consumer, P - provider, B - broker).

The following part presents combinations of these building bricks. An intermediate XWARE instance makes these combinations possible. Communication that deviates from the basic patterns is emphasised in grey colour. Because the patterns are not symmetrical, e.g., from CS to PS is not the reverse of PS to CS, having three interaction paradigms with each two interaction primitives, there are twelve resulting combinations.

CS to PS: Figure 4.16 presents this combination. In Figure 4.16a, a CS consumer (C) initiates a request-response interaction to a PS provider (P). The XWARE instance receives the message and transforms it into a subscribe message before forwarding it to the broker (B). Either there is a matching event at the



(a) Request-response.

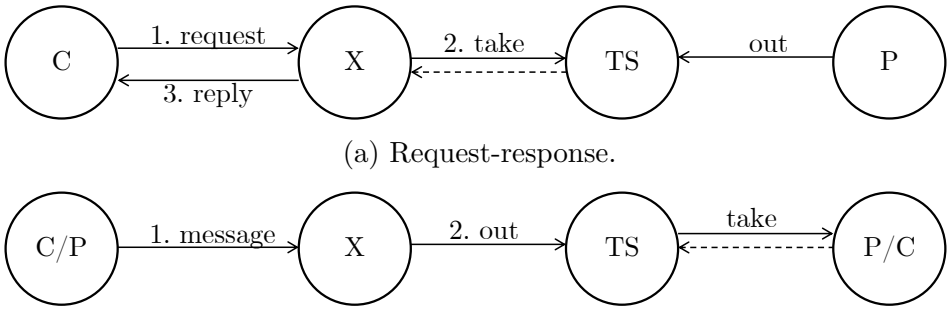


(b) One-way Message.

Figure 4.16.: Interaction Pattern from CS to PS. It is differentiated between the CS primitives request-response (4.16a) and one-way message (4.16b) (C - consumer, P - provider, X - XWARE instance, B - broker).

broker or the PS provider eventually publishes such an event. The broker notifies the XWARE instance about the event. Then, the XWARE instance translates the received publish message into a response message, and delivers it to the consumer. Additionally, the XWARE instance unsubscribes from the broker as only one response is expected. As a matter of fact, this interaction can only be safely executed if the request is either asynchronous or a time out is specified. Otherwise, it can happen that the CS consumer waits for a very long time and, thus, is prevented from doing other computations.

In Figure 4.16b, a CS consumer (or provider) sends out a one-way message to a PS provider (or consumer). A consumer might send a message to a provider that does not expect a response – e.g., a command to turn on the light – or a provider might notify a consumer of something – e.g., a change in the temperature. In the first case, it is a PS provider that needs to subscribe to such events, whereas in the latter case, it is a PS consumer. Regarding the transformation process, the XWARE instance transforms the one-way message into a publish message. After that, it forwards the message to the broker. If a consumer has subscribed for that event, the broker notifies the consumer. Due to the loose time coupling of PS interaction, the last two steps (i.e., subscribe and notify) are independent from the previous ones. That is, these steps can happen before or after the message is published to the broker; therefore, they are are not numbered. Notwithstanding the time of subscription, the consumer will be eventually notified of the event by the broker.



(a) Request-response.

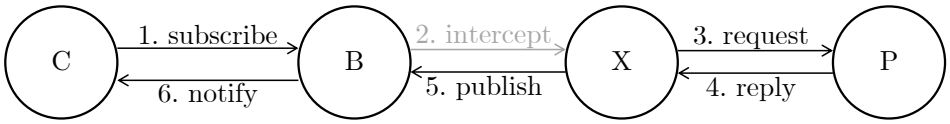
(b) One-way Message.

Figure 4.17.: Interaction Pattern from CS to TS. It is differentiated between the CS primitives request-response (4.17a) and one-way message (4.17b) (C - consumer, P - provider, X - XWARE instance).

CS to TS: Figure 4.17 depicts this combination. In Figure 4.17a, a CS consumer begins a request-response interaction to a TS provider. The request is transformed by the XWARE instance into a take (or in) message. Subsequently, the XWARE instance forwards the message to the TS. Either there is a matching tuple in the TS, or the TS provider eventually writes such a tuple to the TS. The TS sends the tuple to the XWARE instance. There, it is translated into a response message, and sent back to the consumer. As with the combination before, this interaction is only safe if the request-response is performed asynchronously or a time out is specified.

In Figure 4.17b, a CS consumer (or provider) sends out a one-way message to a TS provider (or consumer). The XWARE instance transforms the message into an out message and writes it into the TS. The TS entity reads out the tuple eventually.

PS to CS: Figure 4.18 shows this combination. In Figure 4.18a, a PS consumer subscribes for an event at the broker. The XWARE instance intercepts the message. For the purpose of message interception, either event brokers could be directly executed in the communication plugin, or event brokers often provide an API (see Section 5.2.1). Afterwards, the XWARE instance translates the subscribe message into a request before forwarding it to the CS provider. The CS provider processes the request and sends back a response. The XWARE instance transforms the response into a publish message and delivers it to the broker. Then, the broker notifies the PS consumer. Here, two issues come up: 1)



(a) Subscribe.

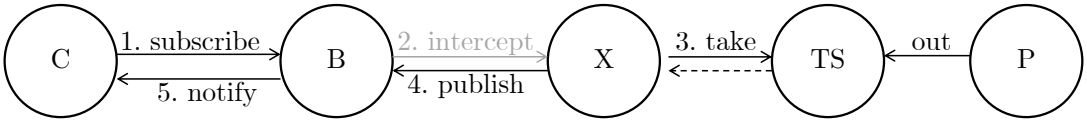
(b) Publish.

Figure 4.18.: Interaction Pattern from PS to CS. It is differentiated between the PS primitives subscribe (4.18a) and publish (4.18b) (C - consumer, P - provider, X - XWARE instance, B - broker).

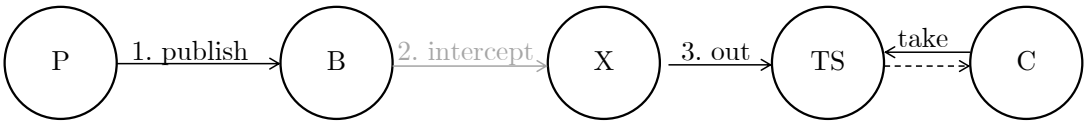
PS consumers subscribe for a time span and not for only one event, and 2) PS consumers subscribe for every event of a specific event type and not from only one provider. Regarding the first matter, the XWARE instance can periodically send out the request. Thus, steps 3 to 6 are executed periodically until the consumer unsubscribes at the broker from the event. With respect to the second issue, which is henceforth referred to as *target selection problem*, there can be different situations at the point of the subscription. First, if there is currently no provider present that provides the required functionality, the XWARE instance needs to wait until there is such a provider. Due to the loose time coupling of PS interactions, this is feasible. Second, if there is exactly one provider that provides the required functionality, the XWARE instance uses it as target. Last, there can be several providers with the required functionality. In this situation, the XWARE instance could send the request periodically to each of those providers, which can lead to a high overhead, but results in an original PS interaction from the point of view of the PS consumer. Another possibility is that the XWARE instance selects a subset (possibly only one) of the available providers. Again, several options exist for selecting a subset of providers, e.g., randomly or the 'best' – however defined. The decision on how to deal with the target selection problem should be left to the developer.

In Figure 4.18b, a PS provider publishes an event that is intercepted by the XWARE instance. There, the event is translated into a one-way message and sent to the CS consumer(s). Again, the target selection problem exists. However,

the reasonableness of this interaction must be raised to question since a publish message should actually only go to entities that are interested in that message, i.e., they have subscribed for it.



(a) Subscribe.



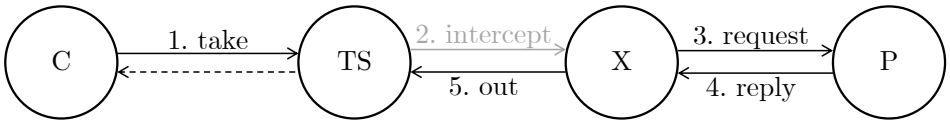
(b) Publish.

Figure 4.19.: Interaction Pattern from PS to TS. It is differentiated between the PS primitives subscribe (4.19a) and publish (4.19b) (C - consumer, P - provider, X - XWARE instance, B - broker).

PS to TS: Figure 4.19 presents this combination. In Figure 4.19a, a PS consumer subscribes for an event at the broker. The XWARE instance intercepts the message, transforms it into a take (or in) message, and sends it to the TS. As soon as the TS provider writes a matching out tuple into the TS, the TS forwards it to the XWARE instance. There, the message is translated into a publish message before it is sent to the broker. Then, the broker notifies the PS consumer of the event. Here again, steps 3 to 5 should be executed periodically until the consumer unsubscribes at the broker from the event.

In Figure 4.19b, a PS provider publishes an event which is intercepted by the XWARE instance. There, the message is transformed into an out message and written into the TS. Eventually, a TS consumer takes the tuple.

TS to CS: Figure 4.20 illustrates this combination. In Figure 4.20a, a TS consumer sends a take (or in) message to the TS. The XWARE instance intercepts this message and transforms it into a request before delivering it to the CS provider. The provider processes the request and sends back the response. On reception, the XWARE instance translates the message into an out message and writes it into the TS. Subsequently, the TS sends the message back to the TS consumer. Here again, the target selection problem occurs and is left to the developer for appropriately dealing with it.

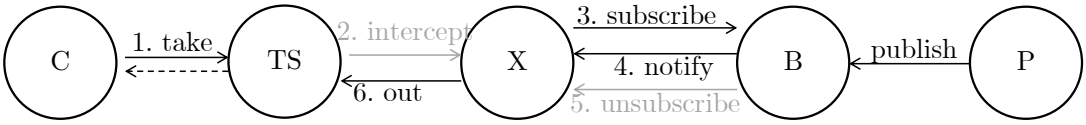


(a) In/Take.

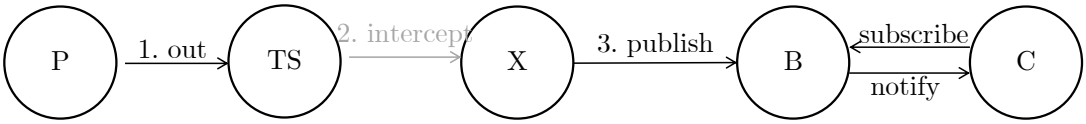
(b) Out.

Figure 4.20.: Interaction Pattern from TS to CS. It is differentiated between the TS primitives in/take (4.20a) and out (4.20b) (C - consumer, P - provider, X - XWARE instance).

In Figure 4.20b, a TS provider writes an out tuple into the TS. The XWARE instance intercepts the message and transforms it into a one-way message which is forwarded to the CS consumer. Here again, the meaningfulness of this interaction has to be questioned, as the out message is usually only read by entities that are interested in the tuple.



(a) In/Take.



(b) Out.

Figure 4.21.: Interaction Pattern from TS to PS. It is differentiated between the TS primitives in/take (4.21a) and out (4.21b) (C - consumer, P - provider, X - XWARE instance, B - broker).

TS to PS: Figure 4.21 depicts this combination. In Figure 4.21a, a TS consumer sends a take (or in) message to the TS. The message is intercepted by the XWARE instance where it is transformed into a subscribe message and sent to the broker. If there is a matching event, the broker notifies the XWARE instance. After that, the XWARE instance translates the event into an out message and forwards it to the TS before unsubscribing from the event at the broker. Then,

the TS sends the tuple back to the consumer.

In Figure 4.21b, a TS provider writes an out tuple into the TS. The XWARE instance intercepts this message, transforms it into a publish message, and sends it to the broker. In case a consumer has subscribed for the event, the broker notifies the consumer.

An appropriate implementation of the interoperability instance permits the preservation of characteristics of the different interaction paradigms. Thus, time coupling, space coupling, and synchronisation coupling can be retained. However, if the consumer works synchronously and accesses a PS or TS provider, it can happen that the consumer has to wait for a very long time for a response.

Summarising, each interaction paradigm might be transformed to each other although the meaningfulness of some combinations can be raised to question (e.g., PS publish to CS or TS out to CS). Furthermore, sometimes the target selection problem exists. Several possibilities have been presented that show how to deal with this problem. Yet, the decision is left to the developer.

These interaction transformations happen in the interaction filter and manage interaction heterogeneity (H3). Having introduced a way in order to align interaction models, the next section focuses on application heterogeneity.

4.5.6. Application Transformation

Application heterogeneity deals with differences in the service operations, such as distinct message and parameter names, or even diverse interfaces. For addressing the first problem, additional extensions are introduced to the XWSDL files. First, `message` elements hold a `maps` attribute. This way, different operation names can be mapped. Second, the `part` element is extended with a `maps` attribute. It indicates that these elements can also be mapped to the according elements of the intermediate definition. This enables a mapping between different parameters. Last, the `part` element may hold a `unit` attribute. This attribute can be used for parameters and results that hold a unit value. As an example, a temperature sensor might return $23°C$ as result. The requester, however, uses a different platform and expects *Kelvin* as unit. If the unit attribute is used, the value is directly converted from *Celsius* to *Kelvin*, preventing result interpretation errors.

The second issue arises from the fact that developers (of smart devices) may implement functionality differently. For instance, an operation op_1 in one platform may be implemented with two operations $op_{2.1}$ and $op_{2.2}$ in another platform. In the following, a personalised service that stores a name serves as illustration. A consumer can change or look up the name. It can happen that, in one platform, there are two operations for setting and getting the full name. However, in another platform, there are operations for setting and getting the first name, and operations for setting and getting the last name. In order to solve this, XWSDL definitions are further extended. First, an additional `maps` attribute extends the `operation` element. Second, the `operation` element allows more than one input and output element now. Thus, one input might be mapped to several inputs and the other way around; the same holds for outputs. Figure 4.22 illustrates these modifications at the example of the name service. There, each `message`, `part` and `operation` element has a `maps` attribute. Further, the `operation` element holds two `input` elements.

```
<definitions name="base.name.IName" maps="SimpleName" >
  <message name="void setFirstname(String)"
              maps="setName" >
    <part name="firstname" type="String" maps="name" />
  </message>
  <message name="void setLastname(String)"
              maps="setName" >
    <part name="lastname" type="String" maps="name" />
  </message>
  ...
  <portType name="SimpleNamePortType" >
    <operation name="setName"
         maps="setNameOperation" >
      <input message="void setFirstname(String)"/>
      <input message="void setLastname(String)" />
    </operation>
    ...
  </portType>
</definitions>
```

Figure 4.22.: XWSDL Example: Name Service. For transforming operations, the maps attributes extend operation elements. Furthermore, several input and output elements are allowed per operation.

Nevertheless, several problems arise here. On the one hand, aggregation and separation of the message content are not trivial and differ for each operation and pair of platforms. A generic solution is very difficult to achieve and could contain further XWSDL extensions, including some regular expression attributes. A simple solution should be preferred. For that reason, the developer has to do the aggregation and separation manually. However, for an adequate support, the building blocks for that are provided. On the other hand, the system has to coordinate discrepancies in the service interfaces. Considering the example in Figure 4.22, the operation should only be sent after both input messages have arrived. Otherwise, the input might be incomplete and, in this case, the full name is overwritten in the target service by only the first (or last) name. Marked petri nets [135] help in solving this issue. The proposed approach differs from the one in [85] in that it considers single methods instead of use cases, and it uses petri nets instead of labelled transition systems.

A *petri net* [135] is a process graph, initially conceptualised for modelling distributed systems. Formally, it can be defined as tuple (S, T, E), where S is a finite set of places, T is a finite set of transitions, and E is a finite set of directed edges connecting either a place with a transition or a transition with a place. A *marked petri net* extends the tuple with M_0, the initial marking. A marking is a set of tokens that is assigned to places. Further, each edge has a weight. The weight w of an edge from a place to a transition indicates that at least w tokens are required in the place in order to fire the transition. The weight w' of an edge from a transition to a place indicates that w' tokens will be produced at the output place if the transition fires. A transition $t \in T$ can fire if each incoming edge can fire. Firing a transition means that tokens are consumed from each input place, and tokens are produced at each output place.

Here, each place represents a transformation state. A transition models the transformation process from one platform representation to another. A petri net is automatically created when an application message arrives. The petri net is built with the help of the `operation` elements in the XWSDL definitions for the source, intermediate, and target platforms. Thus, the starting place represents the arriving message, whereas the final place is the desired representation. If the initiating message expects a response, the response in the source representation is the final place.

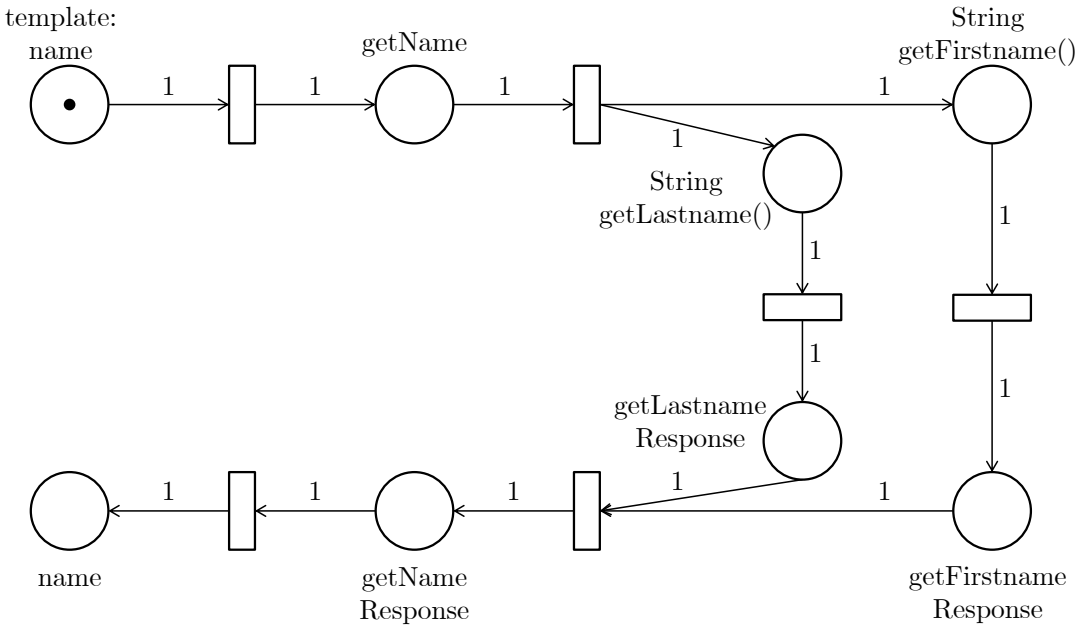


Figure 4.23.: Examplary Petrinet: Get Name. A Limone consumer is asking for the name from a BASE provider.

Figure 4.23 shows an example for a created petri net. There, a Limone-specific consumer intents to get the name from a BASE-specific provider. In Limone, the tuple template is simply `name` to get the full name, whereas in BASE there are two methods for getting the first and the last name. When the Limone application message arrives in the application filter, the petri net is generated. Because the token is at the place that matches the message's operation, the petri net is fired, placing the token in the subsequent place. Also, the message is transformed into the intermediate format, resulting in a `getName` operation. As this matches the place where the token resides, the petri net is fired again. Here, the message is split into two separate messages because the BASE provider has two methods. These messages are sent to the provider, and the petri net is fired. After receiving responses to both of the messages, the net can be fired again, resulting in a state where the `getNameResponse` place holds a token. In this step, the two messages need to be aggregated. The developer is responsible for this aggregation, and therefore, has to write code. After the last transformation, the petri net is in the final state, `getResponse`, and the message is forwarded to the appropriate communication plugin in order to be sent to the actual entity.

Another example (see Figure 4.24) shows a petri net based on the XWSDL

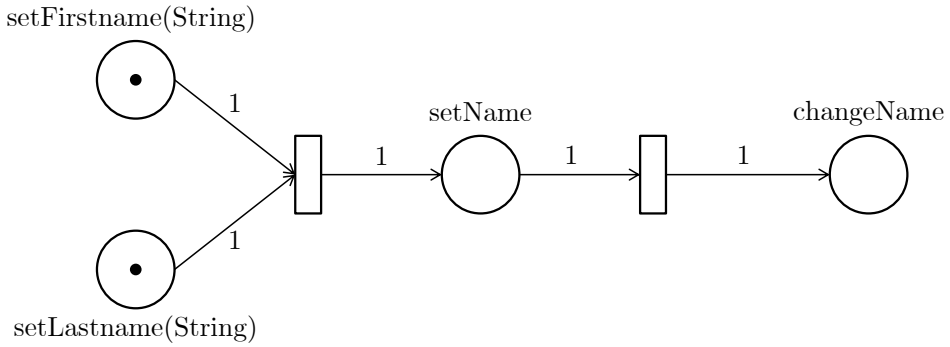


Figure 4.24.: Examplary Petrinet: Set Name. A Limone consumer changes the name at a BASE provider.

definition of Figure 4.22. There, a BASE-specific consumer intents to set the name (or part of it) in a Limone-specific name service. In Limone, the tuple with this intention is simply `changeName`. A problem arises when a message for setting the first (or respectively last) name arrives because there is no information on the last (or respectively first) name. Thus, setting the full name in BASE to only one part of the name makes no sense. Basically, there are two approaches to encounter this problem. On the one hand, it is possible to wait until a request arrives to set the last name and then aggregate the two messages before setting the full name. Another possibility is that the XWARE instance has knowledge about the state of the name (e.g., through requesting it) and takes this information to aggregate the name before forwarding the message. Here, the first option is employed as it goes hand in hand with the petri net approach and does not yield any overhead. However, it has to be taken into account that the name will never be changed if the Limone consumer never sends the second message. Again, the developer is responsible for writing code for the aggregation of the messages.

Overall, application heterogeneity (H5) can be solved with this approach. Furthermore, simple content translations (semantic data heterogeneity, H4) are automatically performed in that data types of parameters can be cast as well as unit conversion takes place. More complex content transformation has be implemented manually using the manual mechanism. For this, a manual transformer has to implement the method `Operations transformOperation(...)`.

These kind of transformations happen in the application filter. The next section points out the transformation of non-functional properties.

4.5.7. Non-functional Properties Transformation

For the non-functional properties heterogeneity, an automatic solution exists for platforms where non-functional properties are described as key-value pairs. Then, a mapping can be performed. In case of a more complex representation of non-functional properties, developers have to use the manual transformation mechanism. For that, the interface of a transformer provides the `Properties transformProperties(...)` method.

In general, there are some problems with non-functional properties. If a platform does not support non-functional properties, one cannot know if it can meet specific properties. Thus, these services are not considered by a request that specifically asks for certain properties. Also, if the sets of non-functional properties for two platforms are not congruent, difficulties can appear. An example is if one platform supports only privacy properties, but a request contains security aspects. Such services should not be considered in the request. Furthermore, the realisation of non-functional properties with an intermediary can be complex. For instance, taking real-time requirements, the communication and transformation by the interoperability instance also takes time. This time must be considered when transforming properties. If a consumer asks for a real-time execution of 200 milliseconds, the time used by the interoperability instance must be deducted when looking for such services. Thus, the interoperability instance itself must include a mechanism for ensuring non-functional properties. For this reason, only the transformation of non-functional properties (H6) is considered, not the adherence to those properties.

There is no separate filter for these transformations, but they are performed when required, e.g., during a service registration. The next section presents notification transformation.

4.5.8. Notification Transformation

Notifications differ in their schemes, content, and delivery modes among platforms. Some of those platforms even do not support notifications. Whereas a filter is responsible for dealing with dissimilarities in notification schemes as well as event contents, the notification management module incorporates mechanisms

to manage different types of delivery modes and non-supporting platforms. The transformations of schemes including the event categories and event content are explained in the following.

Event category transformation must ensure a translation between platform-specific schemes and the intermediate scheme. Therefore, the intermediate language should provide vocabulary for channels, subjects, and content-based systems. A mapping can then be used between the platform-specific and the intermediate vocabulary of the same schemes. Such a mapping could look like 'Climate=Temperature'. If the intermediate vocabulary does not hold a match, either the category could be added to the intermediate vocabulary, or the subscription fails. The first option guarantees completeness. However, it can become unclear depending on the size of the federation. The second option retains a clear pre-defined model. Though, subscriptions might fail and, as a result, notifications are not forwarded. Since usually the administrator defines the federation, this might be desired. In the following, scheme type transformation is explained. At this point, event categories are already in the intermediate language.

Scheme transformation deals with the different forms of PS schemes: channel-based, subject-based, and content-based. For this transformation, a generic solution is desirable. Because content-based notification systems and their data and filter models are highly application-specific, transformation from, to, and between such systems is disregarded in this thesis. However, manual mappings can be specified by developers through code writing.

For the transformations between channel-based and subject-based schemes, the following part describes the transformation model formally and then by way of example of the earlier introduced channels (see Figure 2.6) and subjects (see Figure 2.7). Furthermore, it is assumed that in the intermediate language, channels and subjects are geared to each other name-wise, i.e., if there is a channel *Temperature*, a subject node (as one node in the hierarchy) with the same purpose is named equivalently. Under those circumstances, a mapping between channel names and subject names can be used and works as follows.

Formally, let $C = \{c_1, ..., c_n\}$ be a set of channels and $S = \{s_1, ..., s_m\}$ be a set of subject nodes that build up the subject tree, n is the number of channels, and m is the number of subject nodes. The subject node s_1 is defined as the root of

the subject tree built up by S. A channel $c \in C$ is represented by a channel name, e.g., $c = \mathit{Temperature}$. A subject node $s_l \in S$, $l \in \{1, ..., m\}$, is a vector (l, n_l) where l is the index and n_l is the subject name, e.g., $s_l = (l, \mathit{Light})$. Projections can be used to access the vector values, i.e., $\pi_1(s_l) = l$ and $\pi_2(s_l) = n_l$. A path from the root s_1 to the node s_l through $s_a, ..., s_k$ is represented by $\dot{s}_{1,a,...,k,l} = (s_1, s_a, ..., s_k, s_l)$, where $1 < a < \cdots < k < l \leq m$. Here, due to the nature of subjects, only paths from the root to a subject node are of interest. Further, from the root to each subject node, there exists exactly one path. Therefore, $\dot{s}_{1,a,...,k,l}$ is abbreviated by $\dot{s}_l$ for the convenience of the reader. The auxiliary function $n(\dot{s}_l) = \{s_1, s_a, ..., s_k, s_l\}$ transforms the path $\dot{s}_l$ into the set of subject nodes that make up the path from the root to s_l. Then, let $\dot{S} = \{\dot{s}_1, ..., \dot{s}_m\}$ be the set of all paths starting at the root.

For the mapping from channels to subjects, first, two auxiliary functions are introduced in order to simplify the main formula. Let p be an auxiliary function that maps sets of subject nodes $S' = \{s_a, ..., s_i\} \subseteq S$ to the corresponding sets of paths from the root: $p(S') = \{\dot{s}_a, ..., \dot{s}_i\} \subseteq \dot{S}$. Further, the auxiliary function $q(c) = \{s \in S : \ \pi_2(s) = c\}$, $c \in C$, maps from a channel to all matching subject nodes. Nevertheless, the whole subject paths are necessary here. Thus, the function $m_{channel} = p \circ q$ maps each channel to the corresponding set consisting of all paths to subjects that match the name. In other words, the translation from a channel to subjects consists of traversing the subject tree while checking each subject for a syntactic match. Injectivity and surjectivity of the function $m_{channel}$ depend on the sets C and S. They cannot be assumed though. This can lead to some problems which are pointed out in the following.

If the function $m_{channel}$ maps a channel c to the set $\dot{S}'$, there are three possible outcomes for the number of elements in $\dot{S}'$: 1) $|\dot{S}'| = 0$, i.e., there does not exist a matching subject, 2) $|\dot{S}'| = 1$, i.e., there exists exactly one matching subject, and 3) $|\dot{S}'| > 1$, i.e., there exist several matching subjects. In the first case, the subscription could be neglected, or the event category could be set to the whole hierarchy (by using the wild card to subscribe to the whole sub-tree from the root on). The second option makes sure that no notification is lost. However, it also brings a lot of overhead with it depending on the amount of subjects. Further, consumers may get notifications they are not interested in. As the assumption above indicates that names should be well-matched, it makes

sense to neglect the subscription, despite the fact that some events might get lost. Thus, in the example, the channel *Humidity* would not have a matching subject. In the second case, the matching subject replaces the event category. Considering the exemplary channels and subjects, the channel *Temperature* would match to the subject *PhysicalEnvironment/Conditions/Temperature*. In the third case, a channel name appears more than once in the subject hierarchy. Then, the subscription could be made for all matching subjects or only one; they are possibly chosen randomly or the first one that is found.

For mapping subjects to channels, first, the auxiliary function r is defined as follows: If $\dot{s}_l$ is a subject path and if there are subjects in $n(\dot{s}_l)$ for which there are matching channels in C, then $r(\dot{s}_l)$ is defined as the channel c which matches the subject with the largest index. Due to the fact that subject-based systems often offer the use of wild cards, which can be represented by a set of subject paths, the input of the resulting function needs to be a set of subject paths. Also, as there may be several matches, the output must be a set of channels. If $\dot{S}' = \{\dot{s}_a, \ldots, \dot{s}_i\}$, then the resulting function $m_{subject}(\dot{S}') = \{r(\dot{s}_a), \ldots, r(\dot{s}_i)\}$ matches from a set of subject paths to a set of channels. Put another way, when transforming subjects to channels, for each subject, each subject node on the path to the root is checked against the channel names from the bottom to the root, and the first match is taken. For example, the subject *PhysicalEnvironment/Conditions/Light/Level* would be matched to the channel *Light* because there is no match for *Level*. Furthermore, wild cards may be used. Here, '#' indicates a subscription to a sub-tree, whereas '+' denotes a subscription to the direct children of a subject. When using wild cards, each subject that is included in this subject set is matched. In the example, *PhysicalEnvironment/#* matches to the channels *Light*, *Audio*, and *Temperature*. The subject *PhysicalEnvironment/+* does not match anything. For a more formal description of the mapping from subjects to channels, the interested reader is referred to Appendix C. As above, injectivity and surjectivity of the function depend on the sets C and S, but they cannot be assumed. Problems arising from this are pointed out in the following.

Here again, if the function $m_{subject}$ maps a set of paths $\dot{S}'$ to the set of channels C', there are three possibilities for the number of elements in C': 1) $|C'| = 0$, i.e., there does not exist a matching channel, 2) $|C'| = 1$, i.e., there exists exactly one

matching channel, and 3) $|C'| > 1$, i.e., there exist several matching channels. The addressing of these issues works analogously to the cases above when transforming from channels to subjects.

As a matter of fact, the intermediate scheme should be chosen in a way that no information is lost. Thus, it should use the most expressive scheme type that is used by the platforms in the federation. As until now only a mapping between channels and subjects exist, the subject-based scheme is used as intermediate scheme here.

Because event content is very application-specific, there is no automatic support for translation. Therefore, the `ITransformer` interface provides the `Object transformEvent(...)` operation.

Summarising, the notification transformation deals with part of the notification heterogeneity (H7), i.e., data and scheme. Furthermore, the notification filter is responsible for this task. Having discussed the transformation model, the different filters, and how they address the heterogeneities in detail, the subsequent section gives details on the service management module.

4.6. Service Management

The service management module serves as 'global' service registry and, consequently, stores available services and their descriptions in the abstract semantics and in each supported platform's semantics. As described in Section 4.3.2, instead of translating lookup requests, an XWARE instance only transforms service registrations in order to store them. The translation happens in the alignment module with the help of the discovery filter (see Section 4.5.3). The service management module contains a registry using the intermediate semantics, the *abstract registry*, as well as in each supported platform's semantics, *platform-specific registries*. These different registries accelerate service identifier transformation, service advertisements, and service matching because service descriptions do not have to be transformed every time, but only at their first registration. Moreover, lookup requests can be matched against the transformed platform-specific service descriptions and do not require prior transformation. Thus, each plugin may use a different service matching algorithm.

The overall structure of the service management module is shown in Figure 4.25. The entries differ in the abstract and plugin-specific registries; together, they henceforth are called the *internal registries*. An entry in the abstract registry encompasses the intermediate service identifier (IID) representation as UUID (see Section 4.5.4), the intermediate service description (ISD), and the lease time (or time to live (TTL)). Platform-specific registries hold entries containing the IID, the platform-specific service identifier (ID), and the platform-specific service description (SD). The IID serves as primary key in order to map between different identifier representations among the internal registries. From the TTL value, the abstract registry knows the duration until a service is released, if it is not updated. Therefore, it regularly checks the abstract entries for outdated services and removes them from each internal registry, if necessary (implicit leaving). As some middleware platforms also use an explicit leave mechanism, a service also is removed if a service deregistration comes in. Additionally, a device/registry deregistration message actuates the service management to delete all services that have been advertised from that specific registry. Thus, implicit and explicit leave mechanisms are supported.

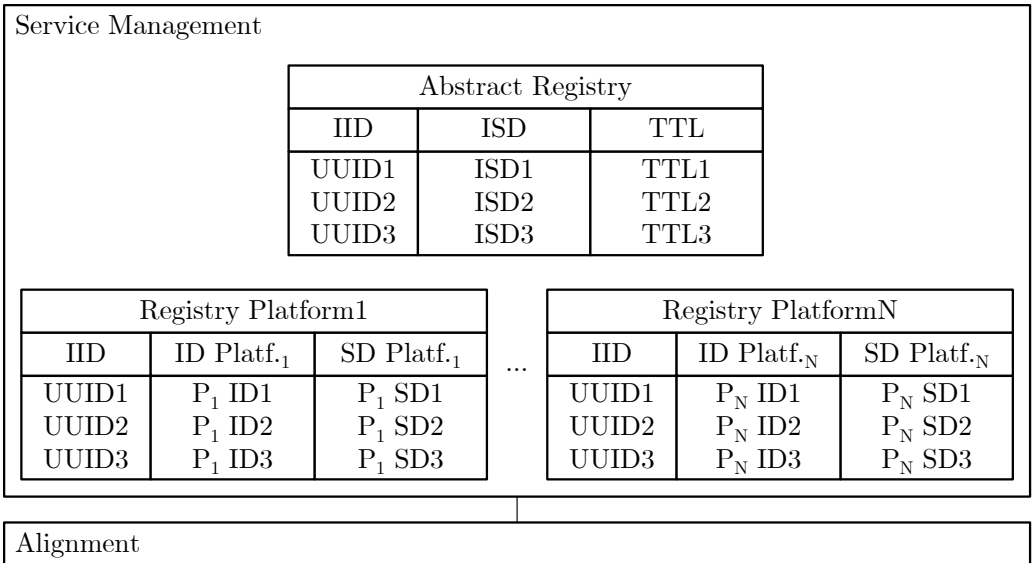


Figure 4.25.: Service Registry. Each internal registry holds IIDs as primary keys to map between different representations ((I)ID - (intermediate) service identifier, (I)SD - (intermediate) service description).

From that, it follows that the service management has to provide several func-

tionalities. When services are registered, they must be added to the internal registries. Further, if a service is already registered, the TTL must be updated. When services deregister or their lease time runs out, they must be removed. For service matching, available services must be received in the platform-specific format. Also, service identifiers must be mapped between the intermediate and platform-specific formats. Furthermore, a service description must be received in a certain representation. Based on these functionalities, the following interface has been defined:

```
interface IServiceMgmt  {
    ServiceDescr addService(ServiceDescr sd, PluginID pid,
        Number ttl);
    void removeService(ServiceID id);
    ServiceDescrs getServices(PluginID pid);
    ServiceID mapServiceID(ServiceID id, PluginID srcPid,
        PluginID tgtPid);
    ServiceDescr getService(ServiceID sid, PluginID pid);
}.
```

So far, the framework's design already allows entities of different platforms to discover and access each other. However, another important functionality is the support of notifications among them. Therefore, the next section explains the architecture and functioning of the notification management module.

4.7. Notification Management

The notification management module, as its name lets assume, is responsible for notification management. In pervasive systems, notifications are usually referring to contextual information, e.g., temperature, light level, or user presence. Several middleware platforms, independent of their interaction paradigm, allow notifications. However, not all platforms support notifications. In order to have a more holistic context view, it is desirable to integrate them as well. In the following, first, the architecture of the notification management module is presented before a mechanism for using non-supporting platforms as providers is introduced. This section is based on [149].

4.7.1. Architecture

Figure 4.26 shows the architecture of the notification management module. It consists of several components derived from the notification management model (see Section 4.3.4): subscription management, provider management, event matching, and storage. At this point, messages are already translated. Thus, they use the intermediate scheme.

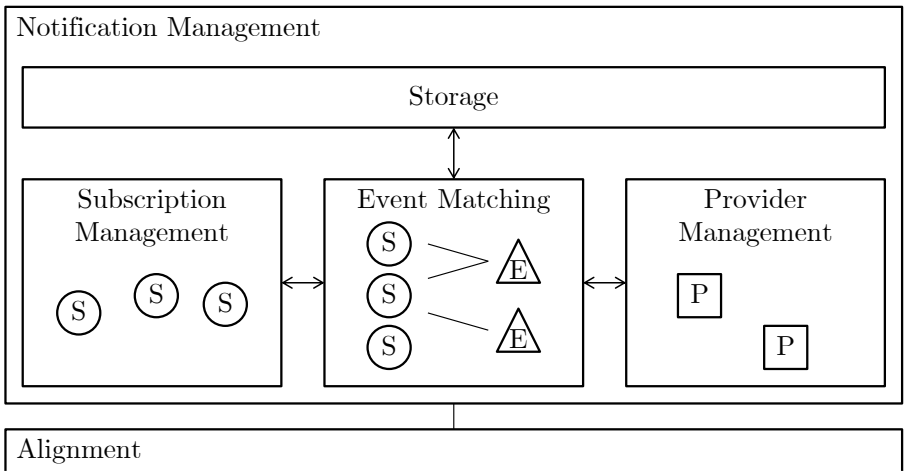


Figure 4.26.: Notification Management Architecture. The notification management module consists of the four components: subscription management, provider management, event matching, and storage (S - subscription, P - provider, E - event).

The *subscription management* component receives and processes subscription messages. Consumers send subscription messages in order to indicate interest in certain events. Such a message requires the consumer's service identifier, an event category or a provider's service identifier, and optionally a lease time. This way, consumers can not only subscribe to an event category but also to all notifications of one provider. For instance, in the UPnP platform, consumers can only subscribe directly to providers. Moreover, a subscription renewal only necessitates the subscription identifier which is returned in response to a successful subscription request. Also, when unsubscribing, the subscription identifier has to be included in order to invalidate the subscription.

The *provider management* component receives or polls new data from providers. Service registration messages that are received by the interoperability instance

are also forwarded to the notification management and the provider management. Furthermore, information on the notification support, used scheme type, as well as the delivery mode have to be known. This knowledge is assumed to be present (e.g., through specification at design time). If it is a supporting platform, the provider management component subscribes to all events from that provider. In case of a provider using the push mechanism, events are automatically received. In case of the pull mechanism, the provider management periodically polls for updates at the provider. Developers can set the time period. If the provider runs on a non-supporting platform, a periodic polling is employed as well (see Section 4.7.2). Incoming events are then forwarded to the event matching component.

The *event matching* component takes incoming notifications and checks them against active subscriptions. This check happens based on the aligned event categories of event and subscription. In case of channels, the matching component performs a syntactic check. In case of subjects, the check is of syntactic nature as well. However, the subscription is inspected for a wild card first. If so, the matching component tests if the event's subject is included in the subscription's sub-tree or tree level, accordingly. If a match is positive, the respective consumer is notified of the event. Besides, incoming events are stored in the storage component.

The *storage* component stores notifications in case that entities request for historical data. Also, administrators may want to analyse notifications. Therefore, the `IStorage` interface provides methods for storing information and for querying stored information. The storage component implements this interface.

Based on the presented architecture, the notification management module provides the following interface:

```
interface INotificationMgmt  {
    void registerProvider(ServDescr service);
    void deregisterProvider(ServiceID provider);
    SubscriptionID subscribe(Object eventCategory,
        ServiceID provider, Number ttl);
    void unsubscribe(SubscriptionID id);
    void publish(ServiceID provider, Event event);
    Events query(Object eventCategory, ServiceID provider);
}.
```

After having introduced the architecture of the notification management module, the following section describes the polling mechanisms for platforms that originally do not support notifications.

4.7.2. Polling for Non-supporting Platforms

From above, one may think that the polling task is quite simple and similar for supporting and non-supporting platforms. However, there is a big difference. With a supporting platform, polling can be performed by including the event category, e.g., *Temperature*, in the poll message. The provider then can easily check if new values exist or return the most recent value. With a non-supporting platform, it is not feasible to use an event category because the platform does not support such a message. Thus, somehow the actual service has to be accessed. For this, the system requires knowledge on the provider with respect to the event category to which the provider can contribute and which method has to be called for this. Thereupon, a further extension to the XWSDL service definition files is introduced incorporating this kind of information. Because at this level the abstract representations are used, it suffices to add the information to the intermediate service definitions.

```
<definitions name="SimpleTemperature"
  sensorType="PhysicalEnvironment/Conditions/Temperature">
  <message name="getTemperature" sensorMethod="true" />
  <message name="getTemperatureResponse" >
    <part name="temperature" type="Double"
        unit="kelvin" />
  </message>
    ...
</definitions>
```

Figure 4.27.: XWSDL Example: Extract of an Intermediate Temperature Sensor. For integrating non-supporting platforms, the `sensorType` and `sensorMethod` attributes further extend the XWSDL file. The newly introduced attributes are underlined.

In order to embody the information in the intermediate service definition files, two new attributes are added there. The `sensorType` attribute extends the `definitions` element. There, the event category is indicated. For instance,

assuming a subject-based scheme, a temperature sensor can hold the sensor type *PhysicalEnvironment/Conditions/Temperature*. Furthermore, the `message` element is expanded with a `sensorMethod` attribute which indicates that this method returns information on the provided event category. These attributes are optional but must only occur once per intermediate service definition. Figure 4.27 shows an extract of an XWSDL file for an intermediate temperature sensor.

Concluding, the notification module manages the remaining part of notification heterogeneity (H7), i.e., support and delivery modes. The next section briefly summarises this chapter.

4.8. Summary

In this chapter, the framework for interoperability between heterogeneous pervasive computing systems, XWARE, has been presented. The framework consists of several abstractions and four modules, namely, communication, alignment, service management, and notification management. Together, they address the whole set of identified heterogeneities, while offering the possibility to extend and customise parts of the framework, including the alignment process. Furthermore, automatic as well as manual transformation specifications are supported.

So far, the concepts and models have been presented. The next chapter introduces the prototype before Chapter 6 evaluates the framework.

5. Prototype

The previous chapter presented XWARE, a general framework for interoperability between different pervasive middleware platforms. This chapter describes the prototype implementation of the framework prior to evaluating it in Chapter 6. First, implementation details are given in Section 5.1. Second, Section 5.2 presents the prototype architecture. Then, Section 5.3 outlines supported platforms and their plugin implementations before Section 5.4 introduces details on the XWARE plugin that enables communication between XWARE instances. Last, Section 5.5 discusses the prototype's limitations.

5.1. Implementation Details

The implementation of the XWARE framework is based on Java, in particular Java Platform, Standard Edition 8 (Java SE 8 [125]). The framework itself does not obligatorily require any additional libraries. However, the implemented storage component used in the notification management module relies on a MySQL database, more specifically the open source database MySQL Server 5.7.16 [126]. Due to the `IStorage` interface (cf. Section 4.7), this can be easily replaced by another storage component. Furthermore, the implemented plugins include dependencies to the platform libraries they provide support for.

Prior to presenting the supported platforms in more detail, the next section describes the overall prototype architecture.

5.2. Prototype Architecture

The current status of the prototype is depicted in Figure 5.1. The architecture complies to the framework's architecture. Each module is implemented in a separate package. Before using the prototype, the administrator has to select the

service definitions, transformers, and domain knowledge, i.e., the XWSDL files or manual transformers, as well as the supported platforms, that should make up the federation. When starting the prototype, the system is configured and the specified components are loaded for the communication module (cf. Section 4.4), alignment module (cf. Section 4.5), service registry module (cf. Section 4.6), and notification management module (cf. Section 4.7). The module implementations are briefly discussed in Section 5.2.1. The prototype, further, includes three additional components, i.e., context component, graphical user interface, and XWSDL generator. Section 5.2.2 briefly introduces those components.

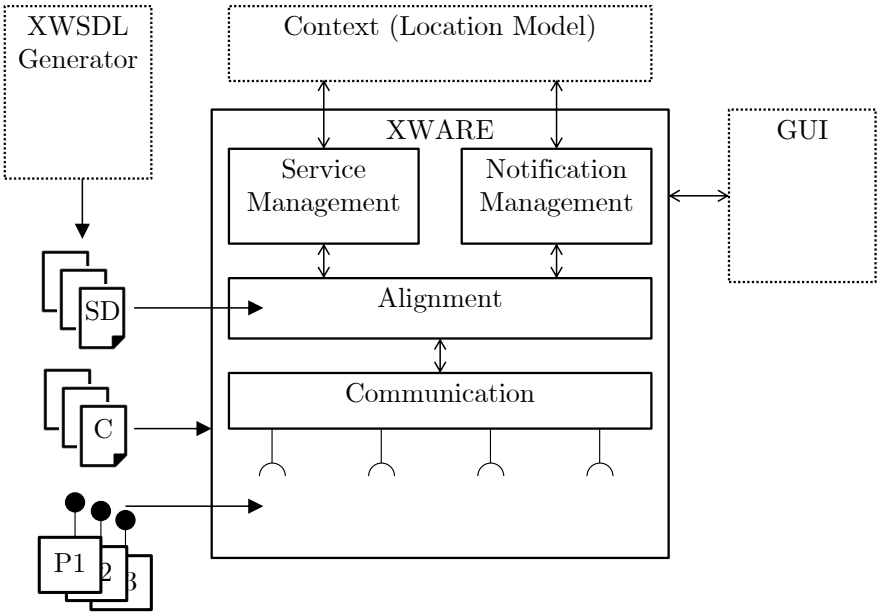


Figure 5.1.: Prototype Overview. The prototype implementation contains the four modules presented in Chapter 4. It further includes a context component and a graphical user interface (GUI). An XWSDL generator supports the developer in specifying XWSDL files. (P - plugin, C - configuration file, SD - service definition).

The prototype uses an event-based architecture [36] for internal communication. An event-based architecture increases decoupling among components through its loose coupling [29, 55]. An event is 'any transient occurrence of a happening of interest' [55]. Components register at an event handler for certain event types. If such an event is passed to the event handler, it is forwarded to registered components which process the event. Here, an event is, e.g., the reception of a message. With respect to the event handler, sometimes two di-

rections of forwarding an event are possible here, e.g., from communication to alignment or from communication to an entity. Therefore, the event handler checks the source of an event and then decides on the recipients. In general, this means that communication that stays within the framework is event-based, whereas communication with external components, i.e., entities, is message-based (cf. [29]). Consequently, at the border of the framework, a conversion must happen from a message to an event at the reception of a message and vice versa when sending a message. This happens in the message converter. For this, the declaration information of the message abstraction is extended by a field that specifies the module that created the event. New components can simply register for event types without any need to change other components. This allows for changes in the architecture without great effort, e.g., addition or replacement of components. Hence, the event-based architecture improves extensibility and customisability with respect to internal components.

In addition, the framework implementation offers reference and skeleton classes for different components, e.g., for advertisement, lookup, or matching. This should ease and speed up development by re-using those components, with or without adjustment. Furthermore, the pre-implemented components base on interfaces. Thus, developers can implement custom components without affecting other parts of the prototype, if necessary. Besides, this interface-based programming approach supports simple configuration of the framework at design time. For this, the prototype uses configuration files. Appendix A shows how such files look like and presents configuration options for a plugin (see Appendix A.1), the alignment module (see Appendix A.2), the service management module (see Appendix A.3), the notification module (see Appendix A.4) and, for the filters (see Appendix A.5).

In the following, further information is given with respect to the main modules.

5.2.1. Modules

The communication module basically consists of zero, one, or several plugins. A plugin incorporates support for a specific platform, and therefore, is responsible for message conversion and communication with entities. The message converter is platform-specific, and hence, requires a custom implementation for each plat-

form. A skeleton component exists that is based on the `IMessageConverter` interface with the intention of supporting developers. Communication between a plugin and entities relies on message interception which is complex by the fact that entities do not know the intermediate instance. Basically, there are two possibilities for this purpose: mimicking communication or using an interception API. For the first option, several reference components exist which developers can re-use, e.g., for the connection manager. The connection manager is responsible for the actual communication with entities. In many platforms, several connections are used for different purposes. Therefore, the connection manager is able to hold several server connections, e.g., for advertisement, lookup, and access, or establish connections as client. Every established connection between the connection manager and an entity is stored with the identifier of the entity and the type of connection. Then, when sending a message, the correct connection is selected through the target identifier together with the message type. In addition, reference and/or skeleton implementations exist for each step of the service discovery model (see Section 4.3.2). The developer can assemble the plugin from those components and/or custom components. Especially for platforms using the CS interaction paradigm, this option works well. In case of TS (and PS) interaction, the plugin has to intercept messages from the TS (and broker respectively). Thus, the plugin has to mimic the TS (and broker respectively). The second option, using an interception API, is only feasible for interaction paradigms that use an intermediate entity, i.e., TS and PS. Then, the intermediate entity automatically forwards each message to the component that implements the interception API – which would be the plugin in this case. However, because messages are directly handed over to the interceptor, time coupling becomes tight. By introducing a mechanism that stores the last events, this disadvantage can be avoided. Several plugins have been implemented in the prototype (see Section 5.3), by employing both of the two options.

The alignment module consists of filters and a repository. The prototype implementation provides a skeleton component for filters. Because filters perform independent tasks, each filter is implemented as a separate thread in view of not blocking the main thread. The following filters have been integrated in the prototype (complying to the framework's design, cf. Section 4.5): discovery, service identifier, interaction, application, and notification (see Figure 5.2).

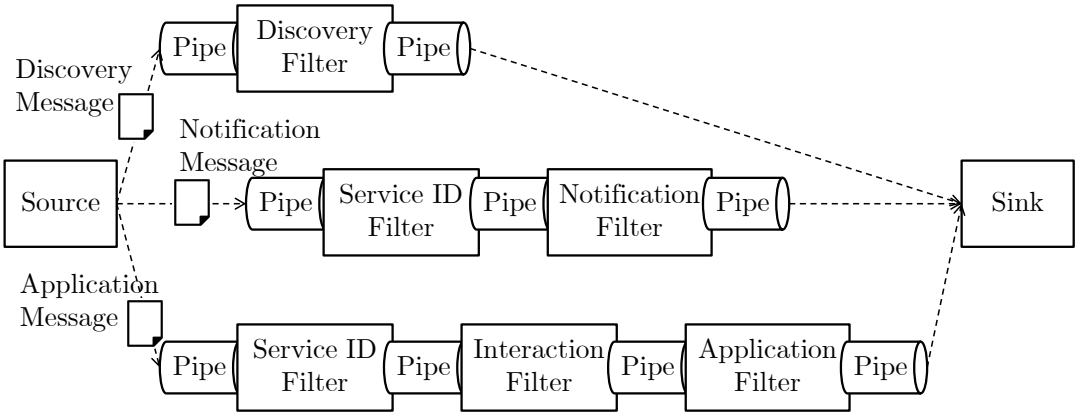


Figure 5.2.: Integrated Filters. A discovery message gets processed by the discovery filter, a notification message by the service identifier (ID) and notification filters, and an application message by the service ID, interaction, and application filters.

The repository stores service definitions, domain knowledge, and (automatic as well as manual) transformers in order to support the filters in their transformation tasks. It reads in this information at start-up. For the manual alignment approach, transformers must be annotated with a `TransformerAnnotation` that allows to specify the platform and functionality of this transformer. At start-up, the repository scans for classes that are annotated by such an annotation. During runtime, the repository automatically selects the manual transformer based on the annotated platform and functionality, if present.

The service management module adheres to its design in Section 4.6. Hence, it stores service descriptions for discovered services in the abstract registry as well as in each platform-specific registry. Furthermore, if a service registration of an already registered service comes in, it checks the description for changes and updates the entries accordingly.

The implementation of the notification management module complies with its design in Section 4.7. Automatic transformation between notification schemes is incorporated for channels and subjects in the notification filter. Regarding a content-based scheme, the developer can manually write code for transformation.

The next section briefly describes additional components that support the developer/administrator or enhance the framework's functionality.

5.2.2. Additional Components

A graphical user interface (*GUI*) has been implemented that keeps track of the available services. For this, the GUI component only needs to register for service registration and deregistration events at the event handler of the service management module.

Additionally, the prototype includes a *context component*. Until now, this component considers the location, and therefore, maintains a location model. The interface of the location model is derived from [11] and allows for position, nearest neighbour, navigation, and range queries. However, the prototype only implements range queries so far. In accordance with the service model (cf. Section 4.3.1), the location is stored as service property.

For an additional support of the developer, an *XWSDL generator* helps in specifying XWSDL files. This generator takes interface files of services as input and produces skeleton XWSDL files where the `maps` and possibly `unit` attributes need to be filled. The generator works for BASE and iCasa services and is easily extensible due to the usage of the strategy design pattern [64].

Having described the prototype architecture, including the module implementations, the following section presents the platforms that are supported by the prototype.

5.3. Supported Platforms

So far, the prototype supports the following platforms: BASE [12], iCasa [99]/iPOJO [50], Cling[1]/UPnP [123], Limone [60], Moquette[2]/MQTT [122], and Redis[3]. In the following, the different platforms and their plugin implementations are briefly outlined.

BASE is a research-based lightweight middleware platform designed for pervasive systems. Service discovery works through device announcements via multicast message. Other entities then can perform service lookups directly on the

[1] https://github.com/4thline/cling
[2] http://andsel.github.io/moquette/
[3] https://redis.io

devices using unicast messages. Service lookup as well as service access are done using asynchronous remote method invocation with proprietary Java objects. BASE uses a proprietary protocol for the interaction between entities and also a specific identifier representation. The plugin implementation re-uses the reference components for service discovery and access. As the message converter is platform-specific it has to be implemented. Furthermore, due to the proprietary communication protocol, the TCP client connection is customised. Two additional classes are required for the identifier representation and its conversion.

iCasa is a pervasive environment simulator using iPOJO as underlying pervasive middleware platform. *iPOJO* is a service-oriented component framework based on OSGi [127]. Communication is performed with Rose [8]. There, a plugin has been implemented that conforms to the reference communication, discovery, and access components. Thus, all components can be re-used here. The only class that must be implemented is the message converter.

Cling is a Java-based implementation of the UPnP protocol stack. *UPnP* is a commercial standard for service discovery and access between services and devices from different manufacturers. Device announcement and lookup requests take place via multicast. Device and service descriptions then are received via an HTTP connection. For this, several reference classes have to be changed, such as multicast, announcement, and HTTP handler. Service invocations are performed via HTTP as well. Furthermore, data is encoded using standards, such as SSDP, HTTP, and SOAP. UPnP supports notifications where one service can register for all notifications of another service. Thus, a notification component needs to be integrated for converting event identifiers. Device descriptions and notifications use proprietary syntaxes based on XML. Therefore, auxiliary classes are used for their conversion. Moreover, the connection manager is changed and the message converter is implemented.

Limone is a research-based middleware platform to ease application development over *ad hoc* networks. It uses the TS interaction paradigm. Each device owns a local TS and a list of remote devices for communication. Devices announce themselves via multicast. Limone does not use a service discovery mechanism. Therefore, a minor change has been made to devices so that they include the names of their services in the announcement message. Without that change, Limone-specific services cannot be used by other entities. Hence, service dis-

covery and access components are re-used. Further, Limone uses a proprietary identifier representation, requiring two additional classes (for the representation and the conversion). Last, the message converter requires implementation.

Moquette is an MQTT-compliant broker for Java. *MQTT* is a connectivity protocol for IoT devices based on the PS interaction mechanism with subject-based event categories. Moquette uses Paho[4] as client implementation. Further, it offers an interceptor interface for its message broker. Thus, messages can be automatically intercepted and processed, i.e., messages are converted into events. Therefore, reference components are not required but only the interceptor is used. Moquette does not use any service discovery mechanism. However, when an event is published, a service is added to the service management module, where the service functionality is derived from the published event category.

Redis is a data structure store that can be used as message broker in order to enable IoT solutions. It uses a channel-based PS interaction mechanism. As client implementation, Jedis[5] is used. Like Moquette, Jedis provides an interceptor interface for its message broker. By nature, Redis has a tight time coupling, i.e., consumer and provider must be available at the same time. However, by the introduced mechanism to store events, loose time coupling is enabled for communication with Redis entities. Jedis does not use any service discovery mechanism. Therefore, the service management module adds a service based on the published event category.

Altogether, the integrated platforms are very diverse. This supports the proposition that the presented XWARE framework is extensible and, furthermore, flexible. Henceforth, the supported platforms are called by their platform/protocol names instead of their specific implementation names, e.g., UPnP instead of Cling. The next section describes the implementation of the XWARE plugin.

5.4. XWARE Plugin

The XWARE plugin enables communication between several XWARE instances. Like platform-specific plugins, the XWARE plugin allows to discover

[4]http://www.eclipse.org/paho/
[5]https://github.com/xetorthio/jedis

services that are registered at other interoperability instances and forward service access or notification messages to other instances. Sent and received messages have to be in the intermediate representation and semantics. The alignment of these messages then happen at the XWARE instances that are connected to the source and target entities.

Reference components of the service discovery model are taken for the service discovery mechanism which works as follows: XWARE plugins periodically multicast their presence. Other XWARE plugins receiving these messages may perform a service lookup for specific or all services at that instance using a unicast message. The reply is also sent as unicast message and includes the matching services.

Furthermore, in *ad hoc* networks, the topology might prevent interoperability instances from communicating directly and, thus, from discovering all available services. Therefore, interoperability instances need to be able to serve as forwarding entity between other instances. As the messages are in the intermediate format here, the XWARE plugin only needs to know the next instance on the route to the target entity. Hence, the topology, or at least the routes between the instances, has to be known by those plugins. For that, the Echo algorithm [38] is used in the prototype in order to create a spanning tree of the interoperability instances. The spanning tree then serves as basis for routing messages to the correct target instance. In order to keep the routing table up to date, the echo algorithm is initiated in the following two cases: an XWARE instance is joining the network, or an instance takes note of another instance's leaving (or crash), possibly due to mobility.

5.5. Limitations

The current prototype has the following limitations. First, the automatic semantic data transformation mechanism, so far, only supports primitive data types and their wrapper classes. Other data types have to be transformed manually.

Second, the prototype supports transformation of non-functional properties, but it does not contain any mechanisms in order to satisfy them. Such mechanisms are very sophisticated, and an intermediary complicates it even more.

This chapter presented the prototype implementation of XWARE. The next chapter showcases the framework's functioning as well as evaluates the XWARE framework on a qualitative and quantitative basis.

6. Evaluation

The previous chapter delineated the prototype of the XWARE framework which is evaluated in the following. Therefore, this chapter concentrates on a proof of concept and a quantitative evaluation. First, a proof of concept showcases the framework's feasibility in Section 6.1 by reference of three realistic use cases. Subsequently, Section 6.2 performs a qualitative requirements assessment before Sections 6.3 and 6.4 do a quantitative evaluation with respect to the overhead for developers and the costs of interoperability. Finally, Section 6.5 discusses the results.

6.1. Proof of Concept

In order to show the feasibility and working of the proposed approach, three use cases have been implemented. The first use case considers shutter management, the second one temperature management, and the third one a smart home with several applications. The use cases are implemented with the iCasa simulator [99]. Therefore, iPOJO entities can be directly added in the simulator, whereas other entities are available through the use of an XWARE instance. In the following, the use cases are explained.

Figure 6.1 depicts the shutter management use case. There is a house consisting of two flats (Alice's and Bob's flats) and one attic. Smart devices are distributed inside and outside of the house, i.e., window shutters are mounted to several windows and photometers are attached near to those windows outside of the house. Each photometer has information on its orientation (north, west, south, or east). As Alice and Bob bought their smart devices independently, they purchased devices of different manufacturers, and thus, using different pervasive platforms. Therefore, they are using XWARE (X) as an interoperability solution to enable their different devices to communicate. Furthermore, in Alice's flat a

shutter management application is executed. This application senses the brightness level at the windows during daytime. Depending on the sensed brightness, the window shutter is moved down or up. One day, the photometer with south orientation in Alice's flat gets broken (1). The shutter management application notices this (2) and wants to use another photometer that has the same orientation. Because there is no such photometer in Alice's flat that uses the same platform, the application asks the XWARE instance for a photometer with south orientation (3). The interoperability solution finds such a sensor in Bob's flat and returns it. Based on that photometer, the application can continue running with a likely similar result (4).

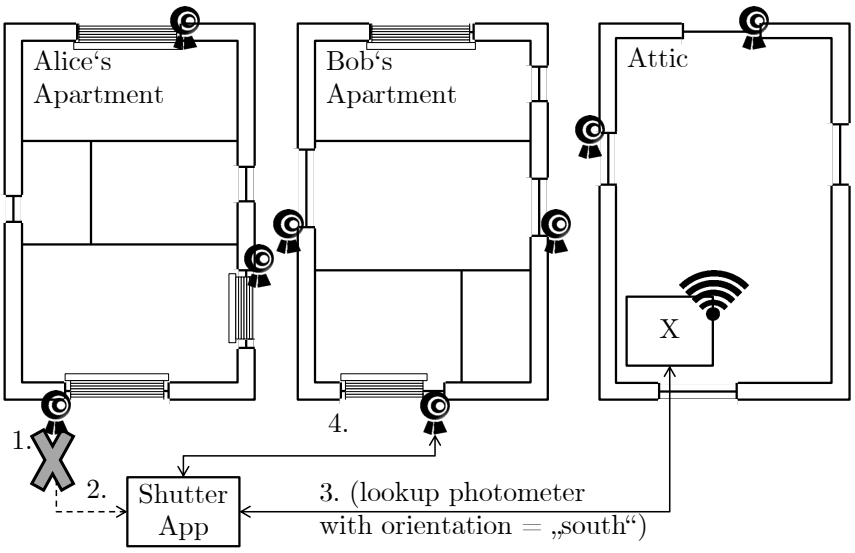


Figure 6.1.: Showcase: Shutter Management. The photometer with south orientation gets broken. The shutter management application (Shutter App), therefore, asks at the interoperability solution (X) for a photometer with the same orientation and uses that one instead.

Figure 6.2 shows the temperature management use case. Again, the use case bases on the house consisting of Alice's and Bob's flats and an attic. Instead of photometers and shutters, thermometers and heaters are distributed over the flats. Furthermore, the XWARE instance (X) includes a location management component. In Bob's flat a temperature management application is executed. This application senses the temperature in the flat and adjusts the heaters accordingly. One day, the thermometer located in Bob's flat gets broken (1). The temperature management application notices this (2) and wants to use another

thermometer. Because there is no further thermometer in Bob's flat, the application asks the XWARE instance for a thermometer in Alice's flat (3), with the help of the location management component. The interoperability solution finds such a sensor in Alice's flat and returns it. Based on that thermometer, the application can continue running (4).

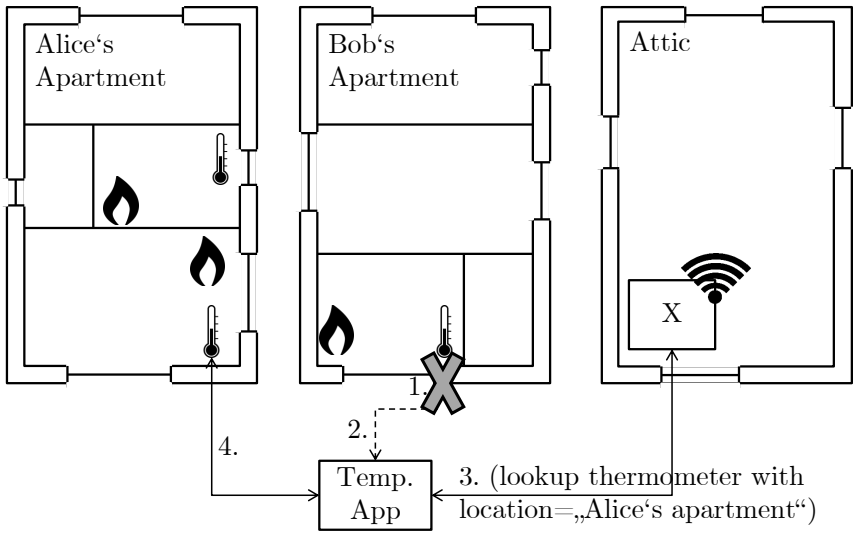


Figure 6.2.: Showcase: Temperature Management. The thermometer in Alice's flat gets broken. The temperature management application (Temp. App), therefore, asks at the interoperability solution (X) for a thermometer located in Bob's flat.

The smart home use case is shown in Figure 6.3. It also bases on the house consisting of Alice's and Bob's flat, and an attic. Furthermore, different devices are distributed that use different middleware platforms, i.e., heaters, window shutters, lights, thermometers, photometers, and presence sensors. Again, an XWARE instance (X) is deployed, including the location management component. Several applications are running in the infrastructure: a shutter management application (see above), a temperature management application (see above), and a light management application. The shutter and temperature management applications work as in the use cases above. The light management application turns the lights in a room on or off depending on whether a person is present.

These use cases show the working, feasibility, and potential of the proposed interoperability approach due to their closeness to reality. The next section performs a qualitative evaluation regarding the requirements.

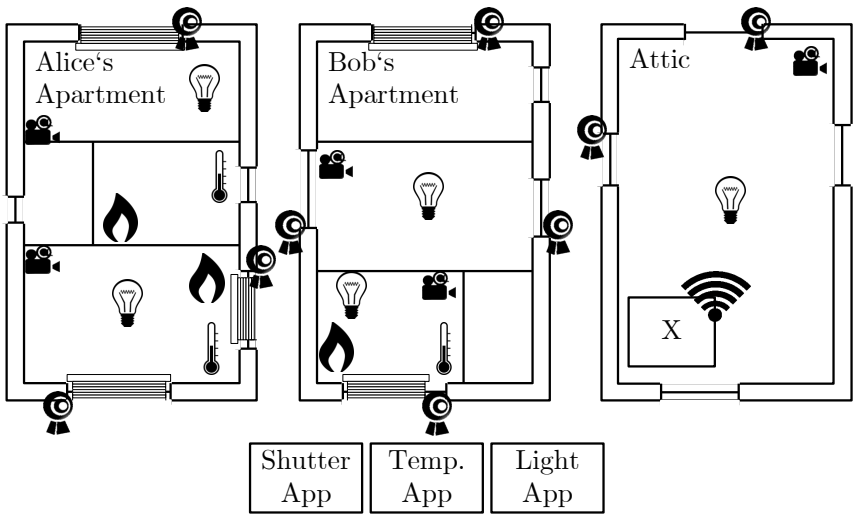


Figure 6.3.: Showcase: Smart Home. Several applications are running in the smart home: a shutter application using photometers and window shutters, a temperature application using thermometers and heaters, and a light application using presence sensors and lights. The devices are running on different platforms, but nevertheless work together through the XWARE instance.

6.2. Requirements Evaluation

After the previous section showed the feasibility of XWARE, this section performs a qualitative evaluation regarding the heterogeneities and requirements.

Regarding the heterogeneities, communication heterogeneity (H1) is tackled by the plugin-based approach of the communication module (cf. Section 4.4). For one platform, there can be several plugins each covering different communication technologies and/or management models. Discovery heterogeneity (H2) is addressed in several locations. First, the general service discovery pattern (cf. Section 4.3.2) and its implemented reference components allow the discovery of services from different platforms without great effort. Second, the service model (cf. Section 4.3.1) abstracts platform-specific service representations and allows a uniform view on services. Third, the alignment module aligns platform-specific service semantics in order to make them available for each supported platform (cf. Section 4.5.3). The framework manages interaction heterogeneity (H3) through the service access abstraction that permits the combination of interaction patterns (cf. Section 4.5.5). Furthermore, difference in the interac-

tion model instantiations are handled in the plugins as they mimic the actual communication. Data heterogeneity (H4) is, due to the fact that it is a general heterogeneity occurring in application, non-functional properties, and notification heterogeneities, solved in the respective locations. Application heterogeneity (H5) is managed by the petri net approach (cf. Section 4.5.6) as well as a mapping of operations and parameters. Non-functional properties heterogeneity (H6) is performed through a mapping between property names (cf. Section 4.5.7). However, there is no mechanism to ensure the adherence of those properties. Last, notification heterogeneity (H7) is addressed through transformation of event categories and schemes (cf. Section 4.5.8), as well as mechanisms to poll data from non-supporting platforms and support different delivery modes (cf. Section 4.7).

With respect to the requirements, the framework is extensible with new platforms due to the plugin-based communication module (cf. Section 4.4). Furthermore, new services can be added either by including XWSDL files or manual specification of transformers. Also, the transformation model (cf. Section 4.5.1) permits to integrate new filters without affecting other ones. Thus, the extensibility requirement (R1) is satisfied. Moreover, the transformation model and the provided interfaces for filters and transformers, as well as the use of configuration files, enables customisation of the alignment process. As well, plugins, service management, and notification management can be easily customised through the provision of interfaces. Therefore, the customisability requirement (R2) is fulfilled. Last, the service management modules keeps track of available services through the use of explicit as well as implicit leave procedures. Hence, the service registry is always up to date, and consequently, the dynamism requirement (R3) is satisfied.

Given these points, the proposed framework covers all of the identified heterogeneities as well as requirements. The next section evaluates the overhead for the integration of new platforms.

6.3. Development Overhead Evaluation

This section describes the overhead for developers in order to integrate a new platform. This is exemplified by the six platforms that have been added to

the prototype so far, namely, BASE [12], iPOJO [50], UPnP [123], Limone [60], MQTT [122], and Redis[1]. The plugin implementations of the different platforms (see Section 5.3) show that the proposed framework is able to include support for a diversity of heterogeneous pervasive and IoT platforms. In the following, the overhead for these plugin implementations is demonstrated.

The overhead is shown in terms of logical lines of code (LLOC). The LLOC metric seems to be an appropriate indicator for the overhead. It measures the lines of code excluding non-statements, such as comments or empty lines. Consequently, it is less susceptible to individual coding styles than other lines of code counting metrics [120]. For the measurement, the CodeCity [2] tool is used.

Table 6.1 shows the number of LLOC for the different plugin implementations. The amount of LLOC the developer needed to write for a plugin ranges from 24 (XWARE) to 1001 (UPnP). It can be noticed that this number is highly correlated with the amount of classes that need to be adjusted and/or added. In other words, the more classes can be re-used from the pre-defined components, the less code needs to be written. The minimum amount of classes that have to be customised are two: plugin and message converter. Furthermore, most code needs to be added for the message converters. It takes between 17 LLOC (84% of the total amount of LLOC for the plugin) for XWARE and 421 LLOC (53%) for BASE. This makes sense as the message converter converts between platform-specific messages and the message abstraction. This is a highly platform-specific task as a differentiation must be made between various messages. Thus, the amount of added code is reasonable. Further, the reason that the plugin for iPOJO is only 25 LLOC is that the communication for iPOJO is based on Rose which is plugin-based. A plugin has been integrated into Rose that uses the same protocol and message formats as the XWARE plugin to not add an additional translation. All in all, the overhead in terms of LLOC is acceptable considering that potentially a great amount of additional services is made accessible by that.

After having implemented a plugin, the developer needs to create the configuration file in order that the plugin can properly start. There, components and further information, e.g., multicast address, have to be specified. Appendix A.1 shows how such a file looks like.

[1] https://redis.io
[2] https://wettel.github.io/codecity.html

Platform	Component	LLOC	Rel. LLOC
BASE		**794**	**100%**
	Plugin	5	1%
	MessageConverter	421	53%
	TCPClientConnection	231	29%
	IDRepresentation	7	1%
	Util	130	16%
iPOJO		**25**	**100%**
	Plugin	4	16%
	MessageConverter	21	84%
UPnP		**1001**	**100%**
	Plugin	14	1%
	MessageConverter	408	41%
	Announcement	16	2%
	Multicast	18	2%
	HttpHandler	103	10%
	Notifications	59	6%
	UPnPDeviceDescription	231	23%
	UPnPNotificationFormat	63	6%
	ConnectionManager	89	9%
Limone		**335**	**100%**
	Plugin	17	5%
	MessageConverter	300	90%
	IDMapper	3	1%
	IDRepresentation	15	4%
Moquette		**88**	**100%**
	Plugin	37	42%
	MessageInterceptor	51	58%
Redis		**142**	**100%**
	Plugin	53	37%
	MessageInterceptor	89	63%
XWARE		**24**	**100%**
	Plugin	8	33%
	MessageHandler	17	67%

Table 6.1.: Development Overhead Evaluation for the Integration of Platforms. Logical Lines of Code (LLOC) are an appropriate metric as they are less susceptible to individual coding styles. The amount of LLOC for adding a new platform seems to be very reasonable regarding the fact that potentially many additional services are made available.

The developer can add XWSDL files to the repository in order to support specific services. Therefore, knowledge about a service is required in the different platforms that should be made interoperable. Appendix B shows XWSDL files for a simple light service in BASE and the intermediate representation. The XWSDL generator facilitates the task of creating these files.

Summarising, the overhead for developers is very feasible regarding the benefit of integrating platforms and services. The next section evaluates the costs for introducing interoperability with XWARE.

6.4. Cost Evaluation

In order to analyse the costs of interoperability, and as a sideline, demonstrate the functioning and feasibility of XWARE, the costs are measured in terms of time (in milliseconds *ms*). For this purpose, three subjects of interest are analysed: 1) service access, 2) inter-instance communication, and 3) notification management.

The measurements were conducted using two notebooks. One notebook has an Intel Core i7-3520M CPU (two cores with 2.9 GHz each), 8 GB of main memory, and is equipped with a 64-bit Windows 7 operating system. The other notebook has an Intel Core i7-5500U CPU (two cores with 2.4 GHz each), 8 GB of main memory, and is equipped with a 64-bit Windows 10 operating system. The two notebooks were connected through a router via Ethernet. For each measurement, every message was sent over the network. Further, every measurement was performed 100 times. For dealing with outliers, the best and worst 5% of the values were excluded, and the average was calculated from the remaining 90 values.

The following sections present the different evaluations and analyses, starting with service access.

6.4.1. Service Access

This section analyses the time to complete an operation, i.e., from sending a message until a response is received. Henceforth, the time to complete an operation is called the *service access time*. The measurement is based on the use case of requesting the light state of a remote light provider. For the baseline,

the service access time is measured between two entities that are using the same platform. Then, in order to analyse the costs of interoperability with the proposed approach, the service access time is measured between every combination of the supported platforms.

Regarding Limone as provider, during the measurements, there was at least one matching tuple in the tuple space at the time of an access in order to make the measurements comparable. Furthermore, in order to make MQTT and Redis measurements feasible, the consumer subscribed 100 times for an event and the time was measured until the first event was received. For this, there was at least one matching event at the time of an access. Due to the introduced mechanism for storing the last published events (see Section 5.3) in the MQTT and Redis plugins, this is possible.

Figure 6.4 shows the evaluation setup for the baseline scenario. There, a consumer (C) is requesting the light state of a provider (P) where both entities are using the same platform. Thus, they can communicate directly without any transformations.

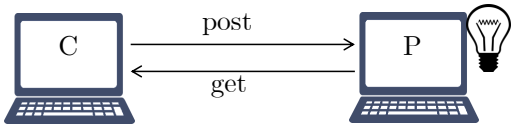


Figure 6.4.: Evaluation Setup: Baseline. A consumer (C) requests the light state from a provider (P). Both entities use the same platform.

Figure 6.5 depicts the logical setup of the evaluation scenario where an interoperability instance is employed in order to enable service access between different platforms. There, the XWARE instance (X) that performs the alignment has to be present.

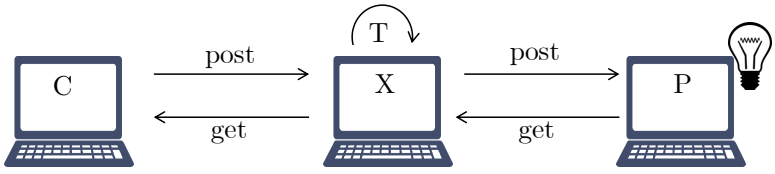


Figure 6.5.: Evaluation Setup: Service Access. Consumer (C) and provider (P) use different platforms. Therefore, an interoperability instance (X) has to perform transformations (T) to enable their interworking.

Table 6.2 shows the results of the service access time measurement for the baseline and for one XWARE instance in *ms*. The grey cells represent the baseline measurement. It can be noted that there are no values for the baseline scenario in the case of iPOJO and Redis. In the case of iPOJO, the reason for that is that the iCasa simulator is used which does not support inter-instance communication. Further, Redis uses PS interaction with a tight time coupling, which is rather unusual. Therefore, if a consumer subscribes at the broker, it will receive the next event that is published. Since this measurement would be very random, it is omitted.

C \ P	**BASE**	**iPOJO**	**UPnP**	**Limone**	**MQTT**	**Redis**
BASE	4.9	8.8	13.4	11.0	4.7	5.7
iPOJO	11.6	-	14.9	17.4	4.7	4.7
UPnP	12.7	14.1	10.2	20.7	8.6	8.9
Limone	17.9	19.0	22.8	11.5	13.7	14.0
MQTT	18.2	19.8	23.0	27.4	2.5	14.3
Redis	10.0	9.8	14.6	22.3	5.5	-

Table 6.2.: Cost Evaluation of Service Access. The time (in *ms*) for completing an operation to receive the light status was measured between every combination of supported platforms. For example, the value 8.8 *ms* (where the BASE row and iPOJO column meet) means that in the use case a BASE consumer (C) requires 8.8 *ms* on average to access an iPOJO provider (P). The grey cells represent the baseline measurement.

The scenario including an interoperability instance for the alignments show, on the one hand, that the prototype works and, on the other hand, that the time for accessing a service is still very feasible. The fastest inter-platform service access requires 4.7 *ms* (on average) and happens between a BASE or iPOJO consumer and a MQTT provider, as well as between an iPOJO consumer and a Redis provider. The most time-consuming service access requires 27.4 *ms* (on average) and happens between an MQTT consumer and a Limone provider. In general, having a maximal service access time of 27.4 *ms* is very appropriate, especially, considering the fact that without the XWARE instance a communication among the platforms would not be possible. Even if users trigger an operation, they would not perceive any delay. Regarding MQTT as consumer, it is noteworthy

that the service access time increases significantly more between intra- and inter-platform communication compared to the other platforms. Due to the fact that the MQTT plugin implementation is quite similar to the Redis plugin implementation, this is a rather surprising finding. The reason for it might be found at MQTT's interception mechanism. Nevertheless, the benefits seem to outbalance the costs after all.

Next, the costs of inter-instance communication are evaluated which allows to increase service availability for interoperability instance with rather minimal configurations.

6.4.2. Inter-instance Communication

This section analyses the overhead of having several XWARE instances with minimal configurations, i.e., one instance supports the consumer's platform, one instance supports the provider's platform, and possibly other instances serve as forwarding entities. Here again, the intra-platform service access time is taken as baseline. The measurement includes every combination of the supported platforms as consumer and provider. The evaluation for Limone, MQTT, and Redis instances works as in the previous measurement.

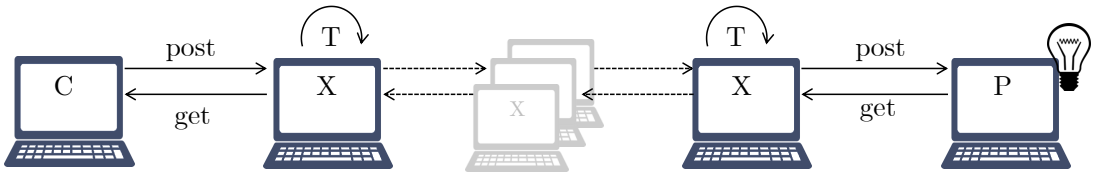


Figure 6.6.: Evaluation Setup: Inter-instance Communication. Consumer (C) and provider (P) use different platforms. XWARE instances (X) perform transformations (T). One interoperability instance supports the consumer's platform, and one instance supports the provider's platform. There may be more instances connecting them.

Figure 6.6 shows the logical setup of the evaluation scenario where several interoperability instances are employed. Especially on mobile devices, XWARE instances might only run a minimal configuration with only one supported platform. Communication between such instances can increase service availability. The translation to the intermediate format takes place on the first XWARE in-

stance, the translation to the target middleware format on the last XWARE instance on the route. Messages in between are in the intermediate format.

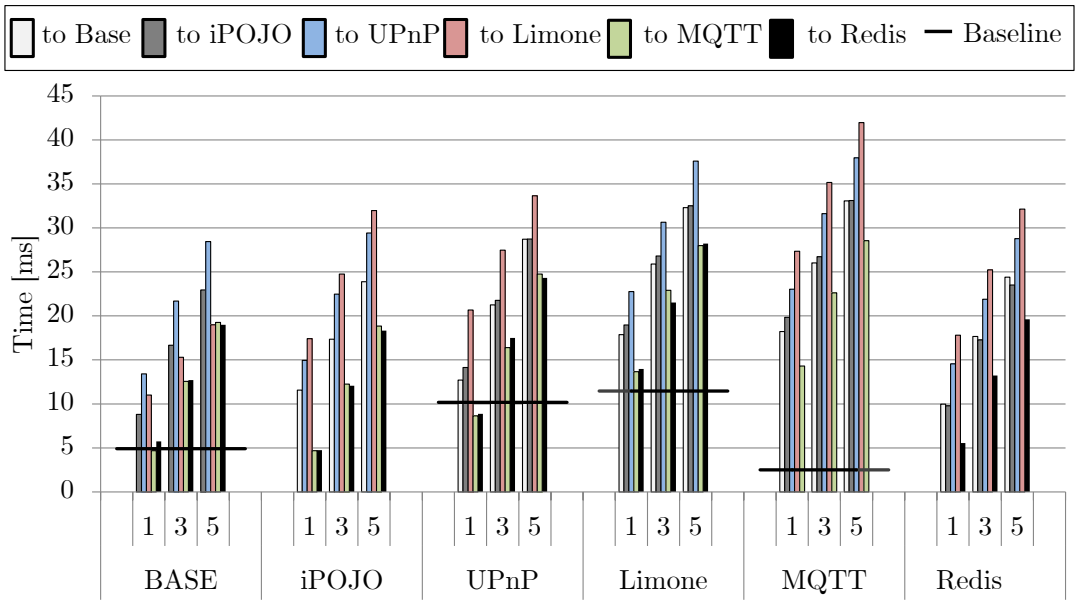


Figure 6.7.: Cost Evaluation of Inter-instance Communication. The service access time (in *ms*) is measured for each combination of supported platforms for one, three, and five interoperability instances in between. The results show an increase of 3.5 *ms* on average in a linear fashion when adding one additional instance.

The results are presented in Figure 6.7 for one, three, and five connected XWARE instances. It depicts the service access time from each platform to every other platform. The x-axis denotes the consumer platform, i.e., the platform that initiates the service access, and the number of interoperability instances. The y-axis shows the service access time. The horizontal lines show the baseline (intra-platform service access) for the different platforms. Per additional XWARE instance in between, the service access time increases in a linear fashion by approximately 3.5 *ms*, independently of the used platforms. The most time, 42.0 *ms* on average, is consumed for a service access from MQTT to Limone with five interoperability instances. The figure also points out the MQTT observation noticed above. Its behaviour is consistent though, narrowing the reason down to a part of the plugin implementation – maybe the interception mechanism. Appendix D shows the concrete result values for three and five interoperability instances. On the whole, the overhead per intermediate node is very acceptable,

especially because without XWARE, inter-platform communication would not be possible.

The next section presents the cost evaluation for the notification management.

6.4.3. Notification Management

For measuring the costs, and testing the feasibility, of the notification management module, the following scenario is used. There is a thermometer that serves as provider. It regularly publishes events of the current temperature. A consumer subscribes for the events of the thermometer for regulating the temperature using a heater. Therefore, the channels and subjects from Figures 2.6 and 2.7 from Section 2.3 are employed. The time is measured for sending an event, i.e., the time from publishing the event until the consumer receives it.

UPnP, Redis, and MQTT originally support notifications (Redis and MQTT can be used as notification system due to their PS interaction style). In the evaluation, a UPnP consumer subscribes to all events from the specific light service, a Redis consumer subscribes to the channel *Temperature*, and an MQTT consumer subscribes to the subject *PhysicalEnvironment/Conditions/Temperature*. Respectively, providers of those platforms publish to the respective consumer/channel/subject. BASE, iPOJO, and Limone do not support notifications.

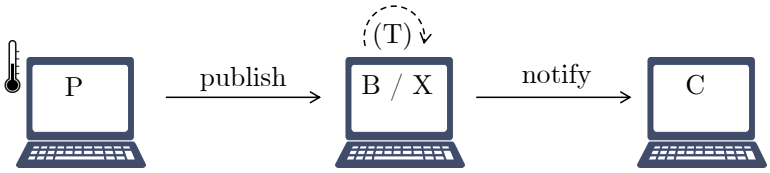


Figure 6.8.: Evaluation Setup: Notification. A provider (P) publishes an event of which a consumer (C) wants to get notified. The event is published to the broker (B). In the baseline scenario, the broker notifies the consumer, while consumer and provider use the same platform. In the case with an XWARE instance (X), the instance intercepts the event, performs translations, and notifies the consumer. Here, consumer and provider use different platforms.

Figure 6.8 shows the logical setup of the evaluation scenario. At this point, the consumer has already subscribed at the broker/interoperability instance. As baseline, the times are measured for direct notifications, i.e., consumer (C) and

provider (P) run on the same platform. As BASE, iPOJO, and Limone do not support notifications, a baseline measurement is not feasible. In the case when an XWARE instance (X) is present, all possible combinations of supported platforms are measured. There, the interoperability instance intercepts the event from the broker (B), translates it, and notifies the consumer.

C \ P	BASE	iPOJO	UPnP	Limone	MQTT	Redis
BASE	-	-	-	-	-	-
iPOJO	-	-	-	-	-	-
UPnP	●	●	5.8	●	13.3	13.3
Limone	-	-	-	-	-	-
MQTT	●	●	19.3	●	5.7	14.4
Redis	●	●	9.7	●	8.1	1.9

Table 6.3.: Cost Evaluation of Notifications. The time (in *ms*) is measured from publishing an event until it is received by the consumer (C). BASE, Limone, and iPOJO cannot publish events, however, the notification management module can poll for new values. '●' denotes the feasibility of this combination, whereas a '-' denotes the infeasibility of this combination. Cells in grey show the baseline.

Table 6.3 shows the results of the measurements. BASE, iPOJO, and Limone cannot be a consumer of events, and therefore, no measurement is possible there. Also, as the notification management module polls values when they are providers, the measurement would not be comparable. Therefore, they are marked with an 'X' to indicate that this combination is possible, but no measurement was performed. A possible solution would be to introduce an event service in those platforms. However, this is a special case and cannot be assumed in every environment, which is the reason for not measuring it. In general, through the use of the proposed notification management module, all platforms can be used as event provider. Further, event category and scheme translation were successfully performed between channels (Redis) and subjects (MQTT). Comparing the measured values with the baseline shows that, using the XWARE instance, sending an event takes up to 7.5 times as much time (from a Redis provider to an MQTT consumer). However, the most consumed time for the sending is only 19.3 *ms* (when sending from a UPnP provider to an MQTT consumer), which is still very fast. Moreover, when considering the fact that without the XWARE

instance notifications would not be possible between different platforms, the benefits outweigh the costs.

Having analysed the costs of the interoperability framework, and thereby also showed its feasibility, the next section discusses the evaluation results.

6.5. Discussion

The goal of the evaluation was to demonstrate the feasibility of the framework's concepts as well as to analyse the costs for introducing interoperability between different platforms. The proof of concept in Section 6.1 showcases the feasibility of the concepts and emphasises the benefits of the proposed framework.

The qualitative requirements evaluation proved that XWARE fulfils all of the elaborated requirements. Indeed, the framework allows developers to extend and customise each module and the alignment process while satisfying the identified set of heterogeneities.

The overhead analysis in Section 6.3 showed that the effort for integrating a new platform is viable. Depending on a platform's adherence to the reference components, more or less lines of code have to be added. Especially when considering the added-value of an integration, the overhead is very reasonable.

Beyond the overhead analysis, the cost evaluation in Section 6.4 demonstrated the feasibility of the framework with respect to the introduced overhead. Although the overhead of a post-get operation is, in relative terms, quite high, the average time does not exceed 27.4 *ms*. Thus, the benefits surpass the costs at least in non-time-critical environments, such as the smart home use case. For time-critical environments, the costs are still acceptable in some cases. However, without the introduction of mechanisms to deal with quality of service properties, a usage in time-critical environments is not recommended.

Altogether, the evaluation of the prototype showed that the concepts of the framework presented in Chapter 4 are suitable for interoperability between pervasive computing systems, not only in theory but also in practice. The next chapter concludes this thesis and gives an outlook on future work.

7. Conclusion and Outlook

The previous chapter evaluated the prototype of the proposed interoperability framework for pervasive computing systems with respect to its working, the satisfaction of requirements, the overhead for integrating new platforms, and the costs of interoperability. This chapter closes this thesis with a conclusion and an outlook on future work.

7.1. Conclusion

The multitude of pervasive platforms nowadays bears several challenges. On the one hand, distinct languages, protocols, and interaction paradigms prevent platforms from interacting with each other. On the other hand, a solution that solves these heterogeneities must satisfy certain requirements, such as extensibility, in order to be able to cope with future platforms and adapt novel alignment algorithms. Therefore, this thesis presented XWARE, an extensible and customisable interoperability framework for pervasive computing systems that supports developers in integrating new platforms, services, and alignment mechanisms.

XWARE's uniform message, service, service discovery, service access, and notification abstractions hide the heterogeneities of distinct platforms. The framework is divided into four functionally independent modules: communication, alignment, service management, and notification management. The communication module is responsible for the actual interaction with entities. Its plugin-based architecture supports the extension of the framework with further platforms. The alignment module performs transformations between messages with respect to semantics, operations, interaction models and properties. For this purpose, the pipes and filters pattern is adopted which enables an easy customisation of functionally independent alignment tasks. Relating to this, the thesis presented an automatic tool that is able to cope with several heterogeneities using XWSDL

files as basis. In addition, the framework includes support for manual transformation specifications, if the automatic tool is not capable. Further services can be integrated by including appropriate service definitions and transformers. Entities joining and leaving the system are monitored with the help of the service management module. The notification management module permits to send and receive notifications across platforms. Additionally, it enables entities to serve as event providers although they do not support notifications naturally, by using a periodic poll mechanism. As a whole, all of the elaborated heterogeneities can be solved.

Finally, the evaluation demonstrated that XWARE is able to enable interaction between a diverse set of pervasive and IoT platforms. There, the costs of interoperability are feasible with respect to the fact that interactions between those platforms would not be possible without XWARE. Furthermore, the overhead evaluation showed that the pre-implemented reference and skeleton components facilitate the integration of new platforms.

7.2. Outlook

During the development of this thesis, several issues have come up that may be worth further investigation.

First, pervasive and IoT applications often require the satisfaction of non-functional properties with respect to contextual or also technical requirements. Nowadays, security and privacy are especially in the focus of pervasive systems and Internet of Things research, e.g., [19] or [156], but other issues are critical as well, such as real-time access. In order to satisfy these requirements, a transformation of their semantics is not sufficient. The integration of a complex framework is required for managing and complying to such non-functional properties.

Second, notification management is performed in a central manner by the prototype. Having several XWARE instances that support notification management may lead to a high overhead as entities receive notifications by several interoperability instances. Therefore, inclusion of a decentralised organisation of notification managers, such as in [137], might be useful.

Third, XWARE was only tested and evaluated in simulated environments until now. A real-world study could further demonstrate the framework's capabilities and reveal additional issues that have not emerged in the simulations.

Fourth, besides the integration of context and conflict management, the next step is to integrate support for cyber-physical systems, such as autonomous driving systems or process control systems. Such systems incorporate not only computational devices and people but also physical processes that influence computations and vice versa [104]. Further, cyber-physical systems are enabled by pervasive computing and Internet of Things. They often provide backend services to users for informational purposes. By integrating plugins and service definitions for these backends into XWARE, such services can be also made available on other platforms.

Bibliography

[1] R. Ahmed, N. Limam, J. Xiao, Y. Iraqi, and R. Boutaba. Resource and Service Discovery in Large-Scale Multi-Domain Networks. *IEEE Communications Surveys & Tutorials*, 9(4):2–30, 2007.

[2] S. Ahmed, M. Sharmin, and S. I. Ahamed. A Smart Meeting Room with Pervasive Computing Technologies. In *Proceedings of International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) and International Workshop on Self-Assembling Wireless Networks (SAWN)*, pages 366–371. IEEE, 2005.

[3] J. Allard, V. Chinta, S. Gundala, and G. Richard. Jini Meets UPnP: An Architecture for Jini/UPnP Interoperability. In *Proceedings of Symposium on Applications and the Internet*, pages 268–275. IEEE, 2003.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley, 1999.

[6] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[7] D. Bandyopadhyay and J. Sen. Internet of Things: Applications and Challenges in Technology and Standardization. *Wireless Personal Communications*, 58(1):49–69, 2011.

[8] J. Bardin, P. Lalanda, and C. Escoffier. Towards an Automatic Integration of Heterogeneous Services and Devices. In *Proceedings of Asia-Pacific Services Computing Conference (APSCC)*, pages 171–178. IEEE, 2010.

[9] L. Baresi, N. Georgantas, K. Hamann, V. Issarny, W. Lamersdorf, A. Metzger, and B. Pernici. Emerging Research Themes in Services-Oriented Sys-

tems. In *Proceedings of Service Research & Innovation Institute (SRII) Summit*, pages 333–342. IEEE, 2012.

[10] A. Bassi and G. Horn. Internet of Things in 2020 - Roadmap for the Future. Technical report, European Commission – Information Society and Media, 2008.

[11] C. Becker and F. Dürr. On location models for ubiquitous computing. *Personal and Ubiquitous Computing*, 9(1):20–31, 2005.

[12] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. BASE – A Micro-broker-based Middleware for Pervasive Computing. In *Proceedings of International Conference on Pervasive Computing and Communications (PerCom)*, pages 443–451. IEEE, 2003.

[13] N. Bencomo, A. Bennaceur, P. Grace, G. Blair, and V. Issarny. The role of models@run.time in supporting on-the-fly interoperability. *Computing*, 95(3):167–190, 2013.

[14] A. Bennaceur. *Dynamic Synthesis of Mediators in Ubiquitous Environments*. PhD thesis, Université Pierre et Marie Curie, 2013.

[15] A. Bennaceur, E. Andriescu, R. S. Cardoso, and V. Issarny. A unifying perspective on protocol mediation: interoperability in the future internet. *Journal of Internet Services and Applications*, 6(1):12, 2015.

[16] A. Bennaceur, G. Blair, F. Chauvel, H. Gang, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, et al. Towards an Architecture for Runtime Interoperability. In *Proceedings of International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 206–220. Springer, 2010.

[17] A. Bertolino, W. Emmerich, P. Inverardi, V. Issarny, F. Liotopoulos, and P. Plaza. PLASTIC: Providing Lightweight & Adaptable Service Technology for Pervasive Information & Communication. In *Proceedings of International Conference on Automated Software Engineering (ASE)*, pages 65–70. IEEE, 2008.

[18] A. Bertolino, P. Inverardi, V. Issarny, A. Sabetta, and R. Spalazzese. On-the-Fly Interoperability through Automated Mediator Synthesis and Monitoring. In *Proceedings of International Symposium On Leveraging Ap-*

plications of Formal Methods, Verification and Validation (ISoLA), pages 251–262. Springer, 2010.

[19] C. Bettini and D. Riboni. Privacy protection in pervasive systems: State of the art and technical challenges. *Pervasive and Mobile Computing*, 17:159–174, 2015.

[20] M. Blackstock and R. Lea. IoT Interoperability: A Hub-based Approach. In *Proceedings of International Conference on Internet of Things (IOT)*, pages 79–84. IEEE, 2014.

[21] G. S. Blair, M. Paolucci, P. Grace, and N. Georgantas. Interoperability in Complex Distributed Systems. In *Proceedings of International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*, pages 1–26. Springer, 2011.

[22] J. Bohn. The Smart Jigsaw Puzzle Assistant: Using RFID Technology for Building Augmented Real-World Games. In *Proceedings of Workshop on Gaming Applications in Pervasive Computing Environments at International Conference of Pervasive Computing (PERVASIVE)*, 2004.

[23] G. Bouloukakis. *Enabling Emergent Mobile Systems in the IoT: from Middleware-layer Communication Interoperability to Associated QoS Analysis*. PhD thesis, Inria de Paris, 2017.

[24] G. Bouloukakis, N. Georgantas, S. Dutta, and V. Issarny. Integration of Heterogeneous Services and Things into Choreographies. In *Proceedings of International Conference on Service-Oriented Computing (ICSOC) Workshops*, pages 184–188. Springer, 2016.

[25] J. Bourcier, A. Diaconescu, P. Lalanda, and J. A. McCann. Autohome: an Autonomic Management Framework for Pervasive Home Applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 6(1):8, 2011.

[26] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple Object Access Protocol (SOAP) 1.1, 2000. W3C Recommendation, Retrieved October 3, 2018 from `http://www.w3.org/TR/SOAP`.

[27] A. Broering, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Kaebisch,

D. Kramer, D. Le Phouc, J. Mitic, D. Anicic, and E. Teniente. Enabling IoT Ecosystems through Platform Interoperability. *IEEE Software*, pages 54–61, 2017.

[28] Y.-D. Bromberg and V. Issarny. Service Discovery Protocol Interoperability in the Mobile Environment. In *Proceedings of International Workshop on Software Engineering and Middleware (SEM)*, pages 64–77. Springer, 2004.

[29] Y.-D. Bromberg and V. Issarny. INDISS: Interoperable Discovery System for Networked Services. In *Proceedings of International Conference on Middleware*, pages 164–183. Springer, ACM/IFIP/USENIX, 2005.

[30] Y.-D. Bromberg, V. Issarny, and P.-G. Raverdy. Interoperability of Service Discovery Protocols: Transparent versus Explicit Approaches. In *Proceedings of Mobile & Wireless Communications Summit*, 2006.

[31] F. Buschmann, R. Meunier, H. Rohnert, and M. Sommerlad. *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley and Sons, 1996.

[32] T. Cai, P. Leach, Y. Gu, Y. Y. Goland, and S. Albright. Simple Service Discovery Protocol/1.0, 1999. IETF Draft, Retrieved October 3, 2018 from `ftp://ftp.pwg.org/pub/pwg/ipp/new_SSDP/draft-cai-ssdp-v1-03.txt`.

[33] M. Caporuscio, P.-G. Raverdy, and V. Issarny. ubiSOAP: A Service-Oriented Middleware for Ubiquitous Networking. *IEEE Transactions on Services Computing*, 5(1):86–98, 2012.

[34] D. Carlson, B. Altakrouri, and A. Schrader. An Ad-hoc Smart Gateway Platform for the Web of Things. In *Proceedings of International Conference on Green Computing and Communications (GreenCom) and Internet of Things (iThings) and Cyber, Physical and Social Computing (CPPSCom)*, pages 619–625. IEEE, 2013.

[35] D. Carlson and A. Schrader. Dynamix: An Open Plug-and-Play Context Framework for Android. In *Proceedings of International Conference on Internet of Things (IOT)*, pages 151–158. IEEE, 2012.

[36] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in Sup-

porting Event-based Architectural Styles. In *Proceedings of International Workshop on Software Architecture*, pages 17–20. ACM, 1998.

[37] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a WideAarea Event Notification Service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.

[38] E. J. H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, 1982.

[39] C. Cho and D. Lee. Survey of Service Discovery Architectures for Mobile Ad hoc Networks. Unpublished Term Paper, Mobile Computing, CEN 5531, University of Florida, 2005.

[40] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web Services Description Language (WSDL) 1.1, 2001. W3C Note, Retrieved October 3, 2018 from `https://www.w3.org/TR/wsdl.html`.

[41] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 160–178. IFIP/ACM, Springer, 2001.

[42] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93, 2002.

[43] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

[44] Dictionary.com. Alignment Definition, 2018. Retrieved October 3, 2018 from `https://www.dictionary.com/browse/alignment`.

[45] W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of International Conference on Mobile Computing and Networking (MobiCom)*, pages 279–286. ACM, 2002.

[46] M. Ehrig. *Ontology Alignment: Bridging the Semantic Gap*, volume 4. Springer, 2006.

[47] C. El Kaed, A. Chazalet, L. Petit, Y. Denneulin, M. Louvel, and F. G. Ottogalli. INSIGHT: Interoperability and Service Management for the Digital

Home. In *Proceedings of Industry Track Workshop at International Conference on Middleware*, pages 3–8. ACM, 2011.

[48] T. Erl. *Service-oriented architecture: concepts, technology, and design.* Pearson, 2005.

[49] C. Escoffier, S. Chollet, and P. Lalanda. Lessons Learned in Building Pervasive Platforms. In *Proceedings of Consumer Communications and Networking Conference (CCNC)*, pages 7–12. IEEE, 2014.

[50] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: An Extensible Service-Oriented Component Framework. In *Proceedings of International Conference on Services Computing (SCC)*, pages 474–481. IEEE, 2007.

[51] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[52] J. Euzenat, P. Shvaiko, et al. *Ontology Matching*, volume 18. Springer, 2007.

[53] M. S. Familiar, J. F. Martínez, and L. Lopez. Pervasive Smart Spaces and Environments: A Service-Oriented Middleware Architecture for Wireless Ad Hoc and Sensor Networks. *International Journal of Distributed Sensor Networks*, 8(4):1–11, 2012.

[54] J. Farrell and H. Lausen. Semantic Annotations for WSDL and XML Schema. *W3C Recommendation*, 2007.

[55] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering Event-Based Systems with Scopes. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 309 – 333. Springer, 2002.

[56] T. Finin, R. Fritzson, D. McKay, et al. A Language and Protocol to Support Intelligent Agent Interoperability. In *Proceedings of National Symposium on Concurrent Engineering (CE & CALS)*, 1992.

[57] C. Flores, P. Grace, and G. S. Blair. SeDiM: A Middleware Framework for Interoperable Service Discovery in Heterogeneous Networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 6(1):6, 2011.

[58] C. A. Flores-Cortés, G. S. Blair, and P. Grace. A Multi-protocol Framework for Ad-hoc Service Discovery. In *Proceedings of International Workshop on*

Middleware for Pervasive and Ad-Hoc Computing (MPAC), pages 10–15. ACM, 2006.

[59] C. A. Flores-Cortés, G. S. Blair, and P. Grace. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. *IEEE Distributed Systems Online*, 8(7):1–11, 2007.

[60] C.-L. Fok, G.-C. Roman, and G. Hackmann. A Lightweight Coordination Middleware for Mobile Computing. In *Proceedings of International Conference on Coordination Models and Languages (COORDINATION)*, pages 135–151. Springer, 2004.

[61] M. Franklin, S. Zdonik, M. Franklin, and S. Zdonik. Data In Your Face: Push Technology in Perspective. In *Proceedings of SIGMOD International Conference on Management of Data*, pages 516–519. ACM, 1998.

[62] A. Friday, N. Davies, N. Wallbank, E. Catterall, and S. Pink. Supporting Service Discovery, Querying and Interaction in Ubiquitous Computing Environments. *Wireless Networks*, 10(6):631–641, 2004.

[63] P. Friess. *Digitising the Industry – Internet of Things Connecting the Physical, Digital and Virtual Worlds*. River Publishers, 2016.

[64] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson, 1995.

[65] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmeja, and K. Wasielewska. Semantic interoperability in the Internet of Things: An overview from the INTER-IoT perspective. *Journal of Network and Computer Applications*, 81:111–124, 2017.

[66] I. Garcia, G. Pedraza, B. Debbabi, P. Lalanda, and C. Hamon. Towards a service mediation framework for dynamic applications. In *Proceedings of Asia-Pacific Services Computing Conference (APSCC)*, pages 3–10. IEEE, 2010.

[67] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[68] N. Georgantas, G. Bouloukakis, S. Beauche, and V. Issarny. Service-Oriented Distributed Applications in the Future Internet: The Case for

Interaction Paradigm Interoperability. In *Proceedings of European Conference on Service-Oriented and Cloud Computing (ESOCC)*, pages 134–148. Springer, 2013.

[69] N. Georgantas, V. Issarny, S. B. Mokhtar, Y.-D. Bromberg, S. Bianco, G. Thomson, P.-G. Raverdy, A. Urbieta, and R. S. Cardoso. Middleware Architecture for Ambient Intelligence in the Networked Home. In *Handbook of Ambient Intelligence and Smart Environments*, pages 1139–1169. Springer, 2010.

[70] I. Gojmerac, P. Reichl, I. P. Žarko, and S. Soursos. Bridging IoT islands: the symbIoTe project. *e & i Elektrotechnik und Informationstechnik*, 133(7):315–318, 2016.

[71] P. Grace, G. S. Blair, and S. Samuel. ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In *Proceedings of Confederated International Conferences "On the Move to Meaningful Internet Systems" (OTM)*, pages 1170–1187. Springer, 2003.

[72] P. Grace, G. S. Blair, and S. Samuel. A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(1):2–14, 2005.

[73] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[74] D. Guinard, V. Trifa, F. Mattern, and E. 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.

[75] E. Guttman. Service Location Protocol: Automatic Discovery of IP Network Services. *IEEE Internet Computing*, 3(4):71–80, 1999.

[76] R. B. Hadj, C. Hamon, S. Chollet, G. Vega, and P. Lalanda. Context-Based Conflict Management in Pervasive Platforms. In *Proceedings of International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 250–255. IEEE, 2017.

[77] S. Helal. Standards for Service Discovery and Delivery. *IEEE Pervasive Computing*, 1(3):95–100, 2002.

[78] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The Gator Tech Smart House: A Programmable Pervasive Space. *Computer*, 38(3):50–60, 2005.

[79] C. Hérault, G. Thomas, and P. Lalanda. Mediation and Enterprise Service Bus: A position paper. In *Proceedings of International Workshop on Mediation in Semantic Web Services (MEDIATE)*, pages 67–80, 2005.

[80] J. Honkola, H. Laine, R. Brown, and O. Tyrkko. Smart-M3 Information Sharing Platform. In *Proceedings of Symposium on Computers and Communications*, pages 1041–1046. IEEE, 2010.

[81] M. N. Huhns and M. P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[82] Hypercat Alliance. Hypercat 1.1 Specification, 2013. Interoperability Action Plan, Retrieved October 3, 2018 from `https://hypercatiot.github.io/spec1.1.pdf`.

[83] IEEE Standards. IEEE Standard Glossary of Software Engineering Terminology, 1990. American National Standard, Retrieved October 3, 2018 from `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=159342`.

[84] IEEE Standards. Interoperability Definition, 2016. Retrieved October 3, 2018 from `https://www.standardsuniversity.org/article/standards-glossary/`.

[85] P. Inverardi, R. Spalazzese, and M. Tivoli. Application-Layer Connector Synthesis. In *Proceedings of International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*, pages 148–190. Springer, 2011.

[86] V. Issarny, A. Bennaceur, and Y.-D. Bromberg. Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In *Proceedings of International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM)*, pages 217–255. Springer, 2011.

[87] V. Issarny, G. Bouloukakis, N. Georgantas, and B. Billet. Revisiting Service-Oriented Architecture for the IoT: A Middleware Perspective. In *Proceedings of International Conference on Service-Oriented Computing (ICSOC)*, pages 3–17. Springer, 2016.

[88] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadist, M. Autili, M. A. Gerosa, and A. B. Hamida. Service-oriented middleware for the future internet: state of the art and research directions. *Journal of Internet Services and Applications*, 2(1):23–45, 2011.

[89] M. Jacoby, A. Antonić, K. Kreiner, R. Łapacz, and J. Pielorz. Semantic Interoperability as Key to IoT Platform Federation. In *Proceedings of International Workshop on Interoperability and Open-Source Solutions*, pages 3–19. Springer, 2016.

[90] A. Kattepur, N. Georgantas, and V. Issarny. QoS Analysis in Heterogeneous Choreography Interactions. In *Proceedings of International Conference on Service-Oriented Computing (ICSOC)*, pages 23–38. Springer, 2013.

[91] J. Kiljander, A. D'elia, F. Morandi, P. Hyttinen, J. Takalo-Mattila, A. Ylisaukko-Oja, J.-P. Soininen, and T. S. Cinotti. Semantic Interoperability Architecture for Pervasive Computing and Internet of Things. *IEEE Access*, 2:856–873, 2014.

[92] T. Koponen and T. Virtanen. A Service Discovery: A Service Broker Approach. In *Proceedings of Hawaii International Conference on System Sciences (HICSS)*, pages 7–13. IEEE, 2004.

[93] N. Koshizuka and K. Sakamura. Ubiquitous ID: Standards for Ubiquitous Computing and the Internet of Things. *IEEE Pervasive Computing*, 9(4):98–101, 2010.

[94] E. Kovacs, M. Bauer, J. Kim, J. Yun, F. Le Gall, and M. Zhao. Standards-Based Worldwide Semantic Interoperability for IoT. *IEEE Communications Magazine*, 54(12):40–46, 2016.

[95] S. Kubler, J. Robert, A. Hefnawy, K. Främling, C. Cherifi, and A. Bouras. Open IoT Ecosystem for Sporting Event Management. *IEEE Access*, 5:7064–7079, 2017.

[96] I. B. Lahmar, H. Mukhtar, and D. Belaid. Interoperability in Pervasive Environments. *Pervasive Communications Handbook*, pages 12.1–12.16, 2010.

[97] P. Lalanda and C. Escoffier. Resource-oriented framework for representing pervasive context. In *Proceedings of International Congress on Internet of Things (ICIOT)*, pages 155–158. IEEE, 2017.

[98] P. Lalanda, E. Gerber-Gaillard, and S. Chollet. Self-Aware Context in Smart Home Pervasive Platforms. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, pages 119–124. IEEE, 2017.

[99] P. Lalanda, C. Hamon, C. Escoffier, and T. Leveque. iCasa, a development and simulation environment for pervasive home applications. In *Proceedings of Consumer Communications and Networking Conference (CCNC)*, pages 1142–1143. IEEE, 2014.

[100] S. S. Lam. Protocol Conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, 1988.

[101] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification, 1999. W3C Recommendation, Retrieved October 3, 2018 from `https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[102] P. Leach, M. Mealling, and R. Salz. RFC 4122: A Universally Unique Identifier (UUID) URN Namespace. Technical report, Network Working Group, 2005.

[103] C. Lee and S. Helal. Protocols for Service Discovery in Dynamic and Mobile Networks. *International Journal of Computer Research*, 11(1):1–12, 2002.

[104] E. A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[105] J.-C. Lee, H.-J. Kim, and S.-H. Kim. Bridging OCF Devices to Legacy IoT Devices. In *Proceedings of International Conference on Information and Communication Technology Convergence (ICTC)*, pages 616–621. IEEE, 2017.

[106] N. Limam, J. Ziembicki, R. Ahmed, Y. Iraqi, D. T. Li, R. Boutaba, and F. Cuervo. OSDA: Open service discovery architecture for efficient cross-

domain service provisioning. *Computer Communications*, 30(3):546–563, 2007.

[107] V. Majuntke, S. VanSyckel, D. Schäfer, C. Krupitzer, G. Schiele, and C. Becker. COMITY: Coordinated Application Adaptation in Multi-Platform Pervasive Systems. In *Proceedings of International Conference on Pervasive Computing and Communications (PerCom)*, pages 11–19. IEEE, 2013.

[108] S. Mann. Wearable Computing: A First Step Toward Personal Imaging. *Computer*, 30(2):25–32, 1997.

[109] C. Marinagi, P. Belsis, and C. Skourlas. New directions for pervasive computing in logistics. *Procedia-Social and Behavioral Sciences*, 73:495–502, 2013.

[110] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, et al. Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, pages 26–42. Springer, 2004.

[111] A. N. Mian, R. Baldoni, and R. Beraldi. A Survey of Service Discovery Protocols in Multihop Mobile Ad Hoc Networks. *IEEE Pervasive Computing*, 8(1):66–74, 2009.

[112] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma. A gap analysis of Internet-of-Things platforms. *Computer Communications*, 89:5–16, 2016.

[113] J. Mineraud and S. Tarkoma. Toward interoperability for the Internet of Things with meta-hubs. Technical report, Cornell University, 2015.

[114] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.

[115] S. B. Mokhtar, D. Preuveneers, N. Georgantas, V. Issarny, and Y. Berbers. EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support. *Journal of Systems and Software*, 81(5):785–808, 2008.

[116] S. B. Mokhtar, P.-G. Raverdy, A. Urbieta, and R. S. Cardoso. Interoperable Semantic and Syntactic Service Discovery for Ambient Computing Environments. *International Journal of Ambient Computing and Intelligence (IJACI)*, 4:13–32, 2010.

[117] K.-D. Moon, Y.-H. Lee, C.-E. Lee, and Y.-S. Son. Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware. *IEEE Transactions on Consumer Electronics*, 51(1):314–318, 2005.

[118] J. Nakazawa, H. Tokuda, W. K. Edwards, and U. Ramachandran. A Bridging Framework for Universal Interoperability in Pervasive Systems. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, page 3. IEEE, 2006.

[119] J. Nakazawa, J. Yura, S. Aoki, M. Ito, K. Takashio, and H. Tokuda. A Description Language for Universal Understandings of Heterogeneous Services in Pervasive Computing. In *Proceedings of International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, pages 161–168. IEEE, 2010.

[120] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Boehm. A SLOC Counting Standard. Technical report, Cocomo II Forum, 2007.

[121] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. OASIS Standard, Retrieved October 3, 2018 from `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[122] OASIS. MQTT Specification Version 3.1.1, 2014. OASIS Draft, Retrieved October 3, 2018 from `http://docs.oasisopen.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`.

[123] Open Connectivity Foundation. UPnP Device Architecture 2.0, 2015. OCF Specification, Retrieved October 3, 2018 from `http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf`.

[124] Open Connectivity Foundation. IoTivity Architecture, 2017. IoTivity Documentation, Retrieved October 3, 2018 from `https://wiki.iotivity.org/architecture`.

[125] Oracle Inc. Java SE at a Glance. Retrieved October 3, 2018 from `https://docs.oracle.com/javase/8/docs/api/index.html`.

[126] Oracle Inc. MySQL - The World's Most Popular Open Source Database. Retrieved October 3, 2018 from `http://www.mysql.com/`.

[127] OSGi Alliance. OSGi Service Platform Core Specification, 2007. Retrieved October 3, 2018 from `https://wiki.searchtechnologies.com/javadoc/osgi/r4.core.pdf`.

[128] F. Paganelli, D. Parlanti, and D. Giuli. A Service-Oriented Framework for distributed heterogeneous Data and System Integration for Continuous Care Networks. In *Proceedings of Consumer Communications and Networking Conference (CCNC)*, pages 1–5. IEEE, 2010.

[129] S. Pantsar-Syväniemi, A. Purhonen, E. Ovaska, J. Kuusijärvi, and A. Evesti. Situation-based and self-adaptive applications for the smart environment. *Journal of Ambient Intelligence and Smart Environments*, 4(6):491–516, 2012.

[130] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of International Conference on Web Information Systems Engineering (WISE)*, pages 3–12. IEEE, 2003.

[131] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, 17(02):223–255, 2008.

[132] H. Park, B. Kim, Y. Ko, and D. Lee. InterX: A Service Interoperability Gateway for Heterogeneous Smart Objects. In *Proceedings of International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 233–238. IEEE, 2011.

[133] S. Park. OCF: A New Open IoT Consortium. In *Proceedings of International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 356–359. IEEE, 2017.

[134] D. J. Patterson, O. Etzioni, D. Fox, and H. Kautz. Intelligent Ubiquitous Computing to Support Alzheimer's Patients: Enabling the Cognitively Disabled. In *Proceedings of International Workshop on Ubiquitous Comput-*

ing for Cognitive Aids (UbiCog) at International Conference on Ubiquitous Computing (UbiComp), page 21, 2002.

[135] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität of Bonn, 1962.

[136] G. Pietro Picco, A. L. Murphy, G.-C. Roman, and G. Pietro. LIME: Linda Meets Mobility. In *Proceedings of International Conference on Software Engineering (ICSE)*, pages 368–177. IEEE, 1999.

[137] P. Pietzuch, D. Eyers, S. Kounev, and B. Shand. Towards a Common API for Publish/Subscribe. In *Proceedings of International Conference on Distributed Event-based Systems (DEBS)*, pages 152–157. ACM, 2007.

[138] C. Preist. A Conceptual Architecture for Semantic Web Services. In *Proceedings of International Semantic Web Conference (ISWC)*, pages 395–409. Springer, 2004.

[139] F. Ramparany, F. G. Marquez, J. Soriano, and T. Elsaleh. Handling smart environment devices, data and services at the semantic level with the FI-WARE core platform. In *Proceedings of International Conference on Big Data (Big Data)*, pages 14–20. IEEE, 2014.

[140] S. Ran. A Model for Web Services Discovery with QoS. *ACM SIGecom exchanges*, 4(1):1–10, 2003.

[141] P. Raverdy, O. Riva, A. de La Chapelle, R. Chibout, and V. Issarny. Efficient Context-aware Service Discovery in Multi-Protocol Pervasive Environments. In *Proceedings of International Conference on Mobile Data Management (MDM)*, pages 3–10. IEEE, 2006.

[142] P.-G. Raverdy, V. Issarny, R. Chibout, and A. de La Chapelle. A Multi-Protocol Approach to Service Discovery and Access in Pervasive Environments. In *Proceedings of International Conference on Mobile and Ubiquitous Systems Workshops*, pages 1–9. IEEE, 2006.

[143] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang. Middleware for pervasive computing: A survey. *Pervasive and Mobile Computing*, 9(2):177–200, 2013.

[144] G. G. Richard. Service Advertisement and Discovery: Enabling Universal Device Cooperation. *IEEE Internet Computing*, 4(5):18–26, 2000.

[145] P. Rodrigues, Y.-D. Bromberg, L. Réveillere, and D. Négru. ZigZag: A Middleware for Service Discovery in Future Internet. In *Proceedings of International Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 208–221. IFIP, 2012.

[146] D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen. The DigiHome Service-Oriented Platform. *Software: Practice and Experience*, 43(10):1205–1218, 2013.

[147] F. M. Roth, C. Becker, G. Vega, and P. Lalanda. XWARE – A customizable interoperability framework for pervasive computing systems. *Pervasive and Mobile Computing*, 47:13–30, 2018.

[148] F. M. Roth, C. Krupitzer, S. Vansyckel, and C. Becker. Nature-Inspired Interference Management in Smart Peer Groups. In *Proceedings of International Conference on Intelligent Environments (IE)*, pages 132–139. IEEE, 2014.

[149] F. M. Roth, M. Pfannemueller, C. Becker, and P. Lalanda. An Interoperable Notification Service for Pervasive Computing. In *Proceedings of International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 842–847. IEEE, 2018.

[150] D. Saha and A. Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *Computer*, 36(3):25–31, 2003.

[151] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[152] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *Proceedings of Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 85–90. IEEE, 1994.

[153] S. Schmid, A. Bröring, D. Kramer, S. Käbisch, A. Zappa, M. Lorenz, Y. Wang, A. Rausch, and L. Gioppo. An Architecture for Interoperable IoT Ecosystems. In *Proceedings of International Workshop on Interoperability and Open-Source Solutions*, pages 39–55. Springer, 2016.

[154] A. Schmidt, M. Beigl, and H. W. Gellersen. There is more to context than location. *Computers & Graphics*, 23(6):893–901, 1999.

[155] H. Shen. Content-Based Publish/Subscribe Systems. In *Handbook of Peer-to-Peer Networking*, pages 1333–1366. Springer, 2010.

[156] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini. Security, privacy and trust in Internet of Things: The road ahead. *Computer Networks*, 76:146–164, 2015.

[157] S. Soursos, I. P. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi, and G. Carrozzo. Towards the Cross-Domain Interoperability of IoT Platforms. In *Proceedings of European COnference on Networks and Communications (EuCNC)*, pages 398–402. IEEE, 2016.

[158] T. Strang and C. Linnhof-Popien. Service Interoperability on Context Level in Ubiquitous Computing Environments. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[159] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé. *Vision and Challenges for Realising the Internet of Things*. European Commission: Information Society and Media, 2010.

[160] Swetina, Jorg and Lu, Guang and Jacobs, Philip and Ennesser, Francois and Song, Jaeseung. Toward a Standardized Common M2M Service Layer Platform: Introduction to oneM2M. *IEEE Wireless Communications*, 21(3):20–26, 2014.

[161] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas. Service Oriented Middleware for the Internet of Things: A Perspective. In *Proceedings of European Conference on a Service-Based Internet*, pages 220–229. Springer, 2011.

[162] G. Thomson, D. Sacchetti, Y.-D. Bromberg, J. Parra, N. Georgantas, and V. Issarny. Amigo Interoperability Framework: Dynamically Integrating Heterogeneous Devices and Services. In *Proceedings of European Conference on Constructing Ambient Intelligence (AmI)*, pages 421–425. Springer, 2008.

[163] A. Tolk. Composable Mission Spaces and M&S Repositories – Applicability of Open Standards. In *Proceedings of Spring Simulation Interoperability Workshop*, 2004.

[164] A. Uribarren, J. Parra, J. P. Uribe, K. Makibar, I. Olalde, and N. Herrasti. Service Oriented Pervasive Applications Based On Interoperable Middleware. In *Proceedings of International Conference on Pervasive Computing (Pervasive) Workshops*, 2006.

[165] S. VanSyckel. *System Support for Proactive Adaptation.* PhD thesis, Universität Mannheim, 2015.

[166] U. Varshney. Pervasive healthcare and wireless health monitoring. *Mobile Networks and Applications*, 12(2-3):113–127, 2007.

[167] H. Vincent, V. Issarny, N. Georgantas, E. Francesquini, A. Goldman, and F. Kon. CHOReOS: Scaling Choreographies for the Internet of the Future. In *Proceedings of International Conference on Middleware Posters and Demos Track*, page 8. ACM, 2010.

[168] T. Wark, P. Corke, P. Sikka, L. Klingbeil, Y. Guo, C. Crossman, P. Valencia, D. Swain, and G. Bishop-Hurley. Transforming Agriculture through Pervasive Wireless Sensor Networks. *IEEE Pervasive Computing*, 6(2):50–57, 2007.

[169] P. Wegner. Interoperability. *ACM Computing Surveys (CSUR)*, 28(1):285–287, 1996.

[170] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, 1991.

[171] C.-L. Wu, C.-F. Liao, and L.-C. Fu. Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology. *IEEE Transactions on Systems, Man, and Cybernetics)*, 37(2):193–205, 2007.

[172] G. Xiao, J. Guo, L. Da Xu, and Z. Gong. User Interoperability With Heterogeneous IoT Devices Through Transformation. *IEEE Transactions on Industrial Informatics*, 10(2):1486–1496, 2014.

[173] F. Zhu, M. W. Mutka, and L. M. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive computing*, 4(4):81–90, 2005.

[174] D. Zuehlke. SmartFactory - Towards a factory-of-things. *Annual Reviews in Control*, 34(1):129–138, 2010.

Appendix

A. Configuration Files

This appendix shows different extracts of configuration files with exemplary information. Those files are used for the start-up of an XWARE instance. For reasons of clarity, the values and types are shortened.

A.1. Plugin

```
SELF_ID=Dummy=...String
SELF_LOCATION=(PROTOCOL://)ADDRESS:PORT(/PATH)=...String
SELF_EXPIRATION=18000=...Long

//connection manager
CM=...connmanager.impl.ConnectionManager=...String
CM_ADVERTISEMENT_CLIENT=...connection.impl.MulticastGroup=...String
CM_LOOKUP_CLIENT=...connection.impl.TCPClientConnection=...String
CM_SERVER_CLIENT=...connection.impl.TCPClientConnection=...String

//advertisement
C_ADVERTISEMENT_SERVER=...connection.impl.MulticastGroup=...String
C_ADVERTISEMENT_ADDRESS=224.12.0.4=...String
C_ADVERTISEMENT_PORT=2238=...Integer

//other
C_SERVER=...connection.impl.TCPServerConnection=...String
C_SERVER_ADDRESS=127.0.0.1=...String
C_SERVER_PORT=55555=...Integer
C_SERVER_CLIENT=...connection.impl.TCPClientConnection=...String

//functions and function-specific options
F_ANNOUNCEMENT=...discovery.impl.SimpleAnnouncement=...String
F_ADVERTISEMENT=...discovery.impl.SimpleAdvertisement=...String
F_INVOCATION=...invocation.impl.SimpleInvocation=...String
F_LOOKUP...discovery.impl.SimpleLookup=...String
F_MATCHING=...discovery.impl.SyntacticMatching=...String
F_ROUTING=...additional.impl.SimpleRouting=...String
F_NOTIFICATION=..additional.impl.SimpleEventing=...String
LEASE_TIMEOUT=10000=...Integer
ADVERTISEMENT_PERIOD=4000=...Integer

//notification scheme (1 channels, 2 subjects, 3 content-based)
```

```
NOTIFICATION_SCHEME=1=...Byte

//message handler
MESSAGE_CONVERTER=DummyConverter=...String

//interaction paradigm (1 CS, 2 PS, 3 TS)
INTERACTION_PARADIGM=1=...Byte
```

A.2. Alignment

```
F_ALIGNMENT=...mediator.impl.SimpleAlignment=...String
F_REPOSITORY=...repository.impl.SimpleRepository=...String
F_FILTERS_FILE=filter.properties=...String
```

A.3. Service Management

```
F_REGISTRY=...registry.impl.SimpleRegistry=...String
REGISTRY_TABLES=...registry.support.SimpleTable=...String
```

A.4. Notification Management

```
F_NOTIFICATION_MGMT=...notification.impl.SimpleNotMgmt=...String
F_NOTIFICATION_STORAGE=...notification.storage.impl.DBStorage=...String
```

A.5. Filters

The following example shows the filter specification for the integrated filters (see Section 5.2). The `filter` tag specifies the filters. The order of the filters is relevant. The `map` tag determines which messages will go through which filter sequence. The numbers here represent certain message types, e.g., 1 is a service discovery message. Filter initialisation information is omitted here.

```
start
filter=DiscoveryFilter
map=1
end

start
filter=ServiceIDFilter,InteractionFilter,ApplicationFilter
map=3,4,12,13,22,23
end

start
filter=ServiceIDFilter,NotificationFilter
map=2
end
```

B. Exemplary XWSDL Files

This part shows two complete XWSDL files for a light service for the intermediate definition (see Section B.1) and the BASE definition (see Section B.2). Extensions to the original WSDL syntax are underlined.

B.1. Intermediate XWSDL File

```
<definitions name="SimpleLight" >
  <message name="setLightState" >
    <part name="value" type="java.lang.Boolean" />
  </message>

  <message name="getLightState"
      resultType="java.lang.Boolean" />

  <message name="getLightStateResponse">
    <part name="result" type="java.lang.Boolean" />
  </message>

  <portType name="SimpleLightPortType">

    <operation name="setLight">
      <input message="setLightState" />
    </operation>

    <operation name="getState">
      <input message="getLightState" />
      <output message="getLightStateResponse" />
    </operation>
```

```
  </portType>
</definitions>
```

B.2. BASE XWSDL File

```
<definitions name="base.light.ILight" maps="SimpleLight">

  <message name="void setState(java.lang.Boolean)"
      maps="setLightState" >
    <part name="arg0" type="java.lang.Boolean"
        maps="value" />
  </message>

  <message name="boolean getState()"
      maps="getLightState" resultType="java.lang.Boolean"/>

  <message name="getStateResponse"
      maps="getLightStateResponse">
    <part name="result" type="java.lang.Boolean"
        maps="result" />
  </message>

  <portType name="SimpleLightPortType">

    <operation name="setLight" maps="setLight">
      <input message="void setState(java.lang.Boolean)" />
    </operation>

    <operation name="getState" maps="getState">
      <input message="boolean getState()" />
      <output message="getStateResponse" />
    </operation>

  </portType>
</definitions>
```

C. Transformation from Subjects to Channels

This section explains the mapping from subjects to channels (cf. 4.3.4) in a formal manner. Let $C = \{c_1, \ldots, c_n\}$ be a set of channels, $S = \{s_1, \ldots, s_m\}$ a set of subjects, and $\dot{S} = \{\dot{s}_1, \ldots, \dot{s}_m\}$ a set of subject paths. The auxiliary function $n : \dot{S} \to \mathcal{P}(S)$ maps from a subject path to the set of subject nodes that build up the subject path ($\mathcal{P}(A)$ denotes the power set of a set A). The auxiliary function $a_1 : \ S \times C \to \{0, 1, \ldots, m\}$ is defined by (for every $s \in S$, $c \in C$):

$$a_1(s, c) = \begin{cases} \pi_1(s) & \pi_2(s) = c \\ 0 & \pi_2(s) \neq c \end{cases}.$$

This means that $a_1(s, c)$ is the index of s if c and s match, and it is 0 if c and s do not match. The auxiliary function $a_2 : \ S \to \{0, 1, \ldots, m\}$ is defined by (for every $s \in S$):

$$a_2(s) = \sum_{c \in C} a_1(s, c).$$

Thus, it holds for every subject $s \in S$:

$$a_2(s) = \begin{cases} \pi_2(s) & \exists c \in C \text{ that matches } s \\ 0 & \neg\exists c \in C \text{ that matches } s \end{cases}.$$

Now, define $i : \dot{S} \to \{0, 1, \ldots, m\}$ (for every $\dot{s} \in \dot{S}$) through

$$i(\dot{s}) = \begin{cases} \max_{s \in n(\dot{s})} a_2(s) & \max_{s \in n(\dot{s})} a_2(s) > 0 \\ 0 & \max_{s \in n(\dot{s})} a_2(s) = 0 \end{cases}.$$

Then, $i(\dot{s})$ is 0 if there is no subject on the path $\dot{s}$ that matches, and $i(\dot{s})$ equals the greatest index of all subjects that match if there exists a subject s on the path $\dot{s}$ which matches a channel. From that, the function $r : \dot{S} \to \mathcal{P}(C)$ can be

defined (for every $\dot{s} \in \dot{S}$) by

$$r(\dot{s}) = \begin{cases} \{\pi_2(s_{i(\dot{s})})\} & i(\dot{s}) > 0 \\ \emptyset & i(\dot{s}) = 0 \end{cases}.$$

Then, $i(\dot{s})$ is the greatest index of a matching subject in $\dot{s}$, $s_{i(\dot{s})}$ is that subject, and $\pi_2(s_{i(\dot{s})})$ is the name of the matching subject, and thus, it is equal to the matching channel. Consequently, the final function $m_{subject} : \mathcal{P}(\dot{S}) \to \mathcal{P}(C)$ can be defined (for every subset of paths $\dot{S}' \subseteq \dot{S}$) by

$$m_{subject}(\dot{S}') = \bigcup_{\dot{s} \in \dot{S}} r(\dot{s}).$$

D. Inter-instance Communication Evaluation Values

In the following, the concrete values of the inter-instance communication evaluation are presented for three XWARE instances (see Table D.1) and five XWARE instances (see Table D.2).

C \ P	BASE	iPOJO	UPnP	Limone	MQTT	Redis
BASE		16.7	21.7	15.3	12.6	12.7
iPOJO	17.4		22.5	24.8	12.3	12.1
UPnP	21.2	21.8		27.5	16.4	17.5
Limone	25.9	26.8	30.6		22.9	21.5
MQTT	26.0	26.7	31.6	35.2		22.6
Redis	17.7	17.3	21.9	25.2	13.2	

Table D.1.: Service Access Time with Three XWARE Instances. (time is denoted in *ms*, C - consumer, P - producer).

C \ P	BASE	iPOJO	UPnP	Limone	MQTT	Redis
BASE		23.0	28.4	19.0	19.3	19.0
iPOJO	23.9		29.4	32.0	18.9	18.3
UPnP	28.7	28.8		33.7	24.8	24.3
Limone	32.3	32.5	37.6		28.0	28.2
MQTT	33.1	33.1	38.0	42.0		28.6
Redis	24.4	23.5	28.8	32.1	19.6	

Table D.2.: Service Access Time with Five XWARE Instances. (time is denoted in *ms*, C - consumer, P - producer).

Publications Contained in This Thesis

- F. M. Roth, C. Becker, G. Vega, and P. Lalanda. XWARE – A customizable interoperability framework for pervasive computing systems. *Pervasive and Mobile Computing*, 47:13–30, 2018.
- F. M. Roth, M. Pfannemüller, C. Becker, and P. Lalanda. An Interoperable Notification Service for Pervasive Computing. In *Proceedings of International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 842–847. IEEE, 2018.
- F. M. Roth, C. Krupitzer, S. VanSyckel, and C. Becker. Nature-Inspired Interference Management in Smart Peer Groups. In *Proceedings of International Conference on Intelligent Environments (IE)*, pages 132–139. IEEE, 2014.

Lebenslauf

Seit 09/2013	Akademischer Mitarbeiter Lehrstuhl für Wirtschaftsinformatik II Universität Mannheim
08/2008 – 07/2013	Bachelor & Master of Science Wirtschaftsinformatik Universität Mannheim