Data-Centric Service-Oriented Management of Things

Marc-Oliver Pahl
Technische Universität München
Email: pahl@net.in.tum.de

***Abstract*—With the Internet of Things, more and more devices become remotely manageable. The amount and heterogeneity of managed devices make the task of implementing management functionality challenging. Future Pervasive Computing scenarios require implementing a plethora of services to provide management functionality. With growing demand on services, reducing the emerging complexity becomes increasingly important. A simple-to-use programming model for implementing complex management scenarios is essential to enable developers to create the growing amount of required management software at high quality.**

The paper presents how data-centric mechanisms, as known from network management, can be utilized to create a service-oriented architecture (SOA) for management services. The resulting shift of complexity from access functionality towards data structures introduces new flexibility and facilitates the programming of management applications significantly. This is evaluated with a user study on the reference implementation.

I. Introduction

The motto of IM2015 is *big data*. The Internet of Things (IoT) is likely becoming an important source for big data. Over their network interfaces, so-called *smart devices* offer access to data sensed from their environment, and allow actuating their environment. Processing data from IoT devices is complex. Two major reasons for the complexity are interface *heterogeneity* and *distribution*. Different smart devices typically provide and accept data via different software interfaces (heterogeneity). Corresponding to their intended functionality, smart devices are typically distributed in a space. The term *smart space* describes a physical space equipped with sensors and actuators that are remotely controlled by software.

The trend of more and more smart devices being available and becoming installed in the real world increases the necessity and complexity of managing them. Related to IoT, the term *management* comprises not only network-related tasks such as preserving optimal connectivity between smart devices, but also implementing user defined higher-level goals such as "energy saving" [1]. The term *smart space orchestration* is introduced to describe such extended management of smart devices via software. An example scenario for smart space orchestration is implementing the management goal "energy saving" in a service. To fulfill its purpose such a service exchanges information with diverse smart devices. It may evaluate data from presence, light, and power consumption sensors to identify energy saving potential. When its logic concludes that the managed smart space is not in use anymore, it switches off all lights by controlling the corresponding actuators.

The need for adding code for communication with distributed heterogeneous devices makes management services complex. Reducing this complexity becomes more important with the growing need for management in existing and especially in novel management domains such as IoT. A prominent example for a domain with urgent need for new solutions to reduce the management complexity is Pervasive Computing. The complexity of smart space orchestration is identified as key challenge that prevents Pervasive Computing from becoming implemented in the real world [2]–[4].

This paper introduces how data-centric service coupling – well known for managing network equipment, e.g. SNMP [5]– can be used to create a Service Oriented Architecture (SOA). The resulting SOA allows decomposing complex management services into smaller, better manageable, and reusable service building blocks. The network management concepts make it simple-to-understand and simple-to-use. This paper uses examples from the IoT and Pervasive Computing domain as smart space orchestration consists of complex management scenarios. However, the introduced concepts are not limited to those domains but usable for implementing complex management tasks in general.

The introduced methods result in a service design that separates application data and access functionality [6]. This separation reduces the implementation complexity for software developers as it allows them to focus on the logic of the management scenario to be implemented instead of designing interface logic that is needed for service interoperability. In addition the separation fosters the reusability of services by making abstract interfaces mandatory for each service. Such reuse reduces the implementation complexity again as only not previously implemented logic has to be developed.

Sec. II identifies requirements on a SOA for management services. Sec. III reviews existing work that also targets modularizing management functionality. Sec. IV introduces the use of data models as abstract service interfaces. Sec. V discusses how the architecture fulfills the identified requirements. The section introduces several advantages of the new SOA design. Sec. VI validates the targeted usability of the approach.

II. Service-Oriented Management – Requirements

To reduce the complexity a service developer perceives, methods that are simple to understand and use are required. The human brain typically solves complex problems by decomposing them into smaller problems that it then solves [7], [8]. The software design equivalent for that methodology are so-called Service-Oriented Architectures (SOA). A SOA

implements methods to combine, reuse, and discover software services to mash-up complex functionality from smaller services [9]. Consequently, a SOA is a suitable methodology for structuring and facilitating smart space orchestration.

This section briefly recaptures relevant requirements on a SOA design (R0-R3) [9], and adds two new requirements that are important for service-oriented smart device management (R4, R5). The presented requirements are referenced throughout the remainder of the paper, e.g. (R1), to illustrate how the presented methodology implements a SOA.

R0 *Compatible Functional Interfaces* are required for service *composition*.

The REST paradigm [10] with its implementation in the World Wide Web illustrates R0.

R1 *Abstract Service Interfaces* are needed to enable *reuse* and discovery of services.

Abstract interfaces separate the interface used to access a functionality from its implementation. Web Services (WS) exemplify this [9].

R2 *Unique Interface Identifiers* (UID) are required to enable *discovery* and reuse of services.

A glossary exemplifies benefits of unique identifiers.

R3 *Service Lookup and Discovery Mechanisms* are needed to enable composition.

The Universal Description, Discovery and Integration (UDDI) of WS is an example for a directory that allows lookup and discovery of services [9].

For being useful to implement complex management functionality, a suitable SOA in addition requires support for low latency service coupling and must be simple-to-use.

R4 *Low Overall Latency* is needed for enabling the decomposition of time-critical management functions.

Sec. V-F presents how the presented design enables computing the expected latency of mashed-up functions.

R5 *Simple-to-use Concepts* are needed for alleviating the implementation of complex management scenarios.

The complexity should result from the logic of a management scenario and not of the methodology to implement it in software. The success of abstract domain specific languages (DSL) such as BPEL [11] underlines this requirement.

III. Related Work

As discussed in Sec. II, modularization is a familiar tool for humans to structure and facilitate complex tasks. Modularization and composition of services are known software architecture concepts. Early general implementations include CORBA [12], or the ANSA trader [13]. A more recent implementation is Web Services (WS) [9].

All three SOA implementations use function-based interfaces (see Sec. IV). This is problematic for interface compatibility (R0) as services can only be composed if they use compatible abstract interfaces. The missing of a global interface directory and convergence mechanisms [14] are problematic (R3) for the reusability of services. Finally the function-based-interface programming methodology is difficult-to-use (R5) as different functionality –implemented in different services– is invoked using different methods. This requires developers to use function calls with diverse signatures in their code.

The previously introduced coupling mechanisms do not provide entity-management-specific support functionality. Therefore management frameworks typically introduce their own service coupling mechanisms. Following, management (middleware) frameworks are reviewed that share our goal of providing high usability for developers. The later work comes from the Pervasive Computing domain since smart space orchestration was identified to require complex management (Sec. I). To make a significant choice among the numerous existing solutions, recent surveys are used [15]–[17]. The work they contain dates back several years. The Pervasive Computing community apparently did not identify relevant new concepts in the meantime [2], [18].

The FOCALE framework [19], [20] shares the premise of this paper that managing complex entities needs to be structured in a modular way via tool support and autonomy. FOCALE introduces abstract concepts for automated execution of business rules (Sec. II) on complex mobile operator networks. It comprises automated, ontology-based integration and management of distributed heterogeneous entities. The semantic transformation between high-level goals and physical network elements is implemented by hierarchically composed management functions [7].

FOCALE and other frameworks such as [21] establish fixed hierarchies to reflect high-level policies to low-level network elements. While maintaining high usability, the methodology proposed in this paper is more flexible by neither assuming a hierarchical (relatively fixed) structure of space orchestration functionality, nor making design choices such as the use of policies, or the support of business rules. It is most likely well suitable for implementing FOCALE.

The CASCADAS framework [22] creates a service-oriented architecture of management (sub-) services similar to this work. It provides dynamic service composition based on predefined policies. Explicit interface modeling (R1) and a service directory (Sec. IV) are missing in CASCADAS.

The CORTEX framework [23] enables dynamic communication between services targeting car-to-car communication. Services consume and produce events exchanged over event channels. Corresponding to our approach, the authors emphasize the facilitating aspects of modularization for service development. In contrast to our work, CASCADAS does not provide explicit abstract service interface definitions (R1).

The JCAF framework [24] implements services in so-called containers. Containers store and automatically synchronize data resulting in loose service coupling. JCAF does not explicitly model service interfaces (R1) nor provide service discovery functionality (R3).

One.World [25] uses event descriptions as abstract service interfaces. A central registry enables services to discover available events and to connect over those events. Like our approach, One.World uses a fixed set of methods to access

variable data structures. Different to One.World, our solution enables a distributed service discovery, supports convergence, and is not limited to events.

A conclusion of this brief review of existing work is that the general approaches bring most flexibility but miss support for facilitating the implementation of complex workflows that are composed of different services. Examples for such services in the energy saving scenario would be gateway services that interface smart devices, reasoning services that derive knowledge from the data provided by (different) gateway services, and orchestration services that implement high level goals such as energy saving by orchestrating the other services. The concrete frameworks that were developed for managing smart spaces instead are in turn too restricted to the workflows their designers had in mind. In addition they typically lack support for reuse of services as they often do neither use abstract service interfaces, nor provide directories for those.

IV. Data Models as Abstract Service Interfaces

In a classical SOA –such as a web service implementation– abstract service interfaces typically consist of method signatures [9]. An example is the interface shown in listing 1.

```
int getTemperature();
void setTemperature(int value);
int getLuminance();
void setLuminance(int value);
```

Listing 1. Classic SOA service interface.

To call a function, typically service instances that implement the defined interface are identified over a directory (see Sec. II). A typical call to obtain data is *[svcInstanceID].getTemperature()*. The addressing of the desired management property happens in three steps: 1) the service instance is identified (discovery), 2) the service instance is addressed, 3) the desired function is addressed.

In network management (NM), information is often exchanged differently between services, using fixed methods and variable data [6]. An example is the Simple Network Management Protocol (SNMP) [5] that offers a fixed set of methods (get, getnext, getbulk, set) to access data of any managed element. The methods to access data of any SNMP agent remain the same. The offered data differ between managed elements. The structures of the varying data are made available via *data models*. Data is stored in data model instances in the Management Information Base (MIB) of a managed element in SNMP.

The functional interface from listing 1 could represent access to the data structure shown in listing 2. Data could be accessed via generic get and set operations, e.g. *[svcInstanceID].get(/temperature)*.

```
o
|-temperature
|-luminance
```

Listing 2. Graphical structure of a managed element's data model.

Both concepts for accessing data are equally powerful as a bidirectional mapping can easily be created by appending the path in the data structure to the command name and vice versa, e.g. temperature ↔ getTemperature(). From a caller's perspective, in the first case the addressing of the targeted information is in the method name, while in the in the second case, it is in the argument. From a developer's perspective, using a fixed method (get) reduces the programming complexity as its semantics are fixed and do not vary for obtaining different management properties.

The principle of shifting functionality into the data addressing can be carried a step further by moving the service instance identifier into the address as well. The resulting call becomes *get(/svcInstanceID/temperature)*. The advantage of this additional shift is that instead of having to discover a service first, the required data can be addressed directly. As each service instance has a unique identifier, each node within its data model instance has a unique address.

Fig. 1 shows a function signature on top, and its mapping to a data model on the bottom left. This data model separates between input and output parameters as the function signature does. This distinction can be mapped to read and write permissions on values of the data model. Therefore it can be omitted as shown on the bottom right.

As an example, the service interface introduced in listing 1 is composed of the two sub nodes for *temperature* and *luminance* as shown in listing 2 in our approach.

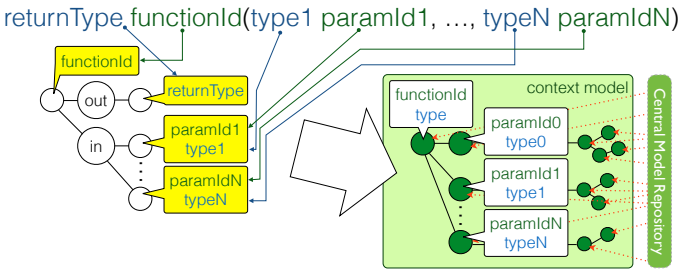

Fig. 1. Mapping between a classic function signature on top and a context model (bottom left) that can then be transformed into the context model on the bottom right as introduced in [14].

A. Data Model Instance Management

In SNMP access to management properties is provided by software agents. SNMP agents are typically device-specific as they interface proprietary functionality. Our design can be implemented in *generic* agent modules that are called *Knowledge Agent* (KA). A KA manages instances of the previously introduced data models for services. A KA can either run stand-alone or be linked to a service.

In the reference implementation of our design the KAs run as stand-alone components. This has the advantage that they can provide access to the knowledge of a service independent of the service running.

In both cases (module or stand-alone), the KAs can fully manage the access to data model instance nodes including security and remote access. The data node management includes authorization of data node accesses and authentication of services. Though not covered in this paper, this can be done using access groups for read and write access for instance. In addition the KAs handle data exchange with remote KAs that manage the data of other services. The resulting architecture is

a secure content-centric network [26]. The content addresses are of the type *knowledgeAgentId/serviceId/pathToValue* [14].

Using the KAs enables developers to fully concentrate on the implementation of the scenario logic [6].

B. Fixed Functions to Access Data Model Instances

A data model only structures the data that is exposed by a service. As presented before with the mapping (Fig. 1), in our approach the data model models the interface of a service. When a service gets started its corresponding data model gets initialized by the KA. Service interaction happens then by accessing each others data model instances via the local KAs. The supported coupling modes are defined by the generic interface methods offered to access the data nodes. By analyzing many scenarios, the following set of functions to access nodes in data model instances emerged. Those methods are fixed and used by each service in our architecture.

- **get**(address), to *read* a value from a node within a data model instance.
- **set**(address, value), to *write* a value to a node within a data model instance.
- **(un-)lock/rollback**(address) to lock a subtree (Sec. IV) of a data model instance for exclusive use.
- **(un-)subscribe**(address, callback), to receive a change notification for a value in a data model instance.
- **(un-)registerVirtualNode**(address, callback), to receive a callback on a value change at *address*.

Get and *set* are used to read and write data values. They implement blackboard communication over the KAs (Fig. 2) [27], [28].

Since properties of management interfaces often have to be set in an atomic way (e.g. network address and network mask [20]), transactions are necessary (*(un-)lock/rollback*). The introduced hierarchical data model facilitates transactions as it allows to model correlated properties in a common subtree that can then be locked for exclusive access. In addition locking any set of nodes must be supported for the diverse use cases one can think of. As an example a service that opens the garage door may want to gain exclusive access to automated things in front of the door before opening it so that no other service moves them to the door while opening.

Triggering another service is a very common interaction in management. In case of smart space orchestration the initially described energy saving action of switching the lights off would typically be implemented by triggering the gateways controlling the actual lights. More concrete the managing service would set the node containing the state of each light to off (e.g. *set /lightId/isOn 0*). To enable coupling on such events, a subscription on a value change is supported on each subtree (including leafs). Whenever a node's value changes, a notification is sent to all subscribers. This enables loose coupling.

The previously described functionality is (partly) found in most management architectures today [29]. A problem of using data structures for service coupling is the typically missing possibility of direct (temporal and referential) service coupling. Therefore a novel coupling mode is introduced, so-called *virtual nodes*. Accessing a virtual node is fully transparent for a caller service but it results in an immediate service invocation at the called service. As a concrete example, in case of being implemented as virtual node, requesting the value of the temperature node in listing 2 results in a computation of which the value is immediately returned to the caller. This mechanism may appear simple and yet it turns out to be powerful as shown in Sec. V.

V. A Service-oriented Management Architecture

This section discusses the SOA properties (Sec. II) of the introduced architecture (Sec. IV).

The introduced service coupling methodology results in a fundamental mental change of service invocation. Classic SOA interfaces implement procedural thinking. Computational processes are identified and invoked as functions. The proposed use of data models as abstract service interfaces shifts the coupling of services towards a *descriptive* task. Information – represented as data– is identified and accessed not the construct (function) that produces it. The coupling of services becomes linking data – not functions.

Descriptive interfaces are simpler-to-understand and use (R5) for humans than procedural ones [8]. Describing the properties that a managed entity offers is more straight forward than defining message signatures. It does not require knowledge about functional programming. Instead it is based in semantic modeling where entities of the real world are represented as *virtual objects* [14], [30], [31].

A. Service Interface Modeling

Using data models for services requires the definition of a data model for each service. In [14] we introduced mechanisms for intuitive data modeling and crowdsourced convergence of the resulting models. To summarize the important points for this paper, a *Central Model Repository* (CMR) was introduced that contains all data models that were defined with a unique ID (UID) each (R2). The UIDs in Fig. 1 are *functionId*, *paramId0*, ..., *paramIdN*. The CMR implements a type system where data models are the types, identified by their UID.

Data models can be composed to complex types as shown on the rights of Fig.1 where *type* consists of sub nodes with one type each (*type0*, ..., *typeN*). The types are again composed of nodes of other types that are available from the CMR. In the end, each node inherits from one of the few basic data types (*text*, *number*, *list*). This inheritance is relevant for validation of values and for interface compatibility (Sec. V-C).

Our approach makes abstract service interfaces mandatory (R1). It does not result in additional implementation work as the data structures to provide the data a service exposes have to be designed in any case (R5). At the same time the data modeling process is more intuitive than the definition of function signature based interfaces.

As novel aspect that is typically missing in today's SOAs, the CMR provides functionality to converge abstract service interface descriptions (data models) (R0). Interface convergence is especially critical in the IoT domain where interoperability

is required to implement complex scenarios in different smart spaces (service portability). As a concrete example the energy saving service targets all lights independent of which kind and from which vendor they are. Consequently all gateways to lights should use compatible interfaces (data models) [14].

Finally the CMR acts as directory for abstract service interfaces (data models). Developers can look up the type identifiers that describe the data they require at run time to provide the desired functionality of the service they implement (R3).

B. Service Discovery

The discovery of services becomes the discovery of data. This is intuitive as the logic of a service requires certain input and output data. Each data model in the CMR has a UID that can be used for identification (R2). At any time in the life-cycle of a service (e.g. development time, run time), other services can be referenced using their type UID. As an example, the energy saving service can search for data nodes of type "/lighting/light". Its KA returns all instance addresses of data nodes that have this type (R3). The type-identifier-based discovery searches not only the root but all levels of a data model.

Using the type identifier for search to identify service instances implements a locator-id-split [32]. The type is used to identify data. The location of services offering the data can remain transparent. This is important for dynamic environments such as smart spaces where devices are added and removed at run time. In the energy saving example new lights may occur but can still be switched by the service as they are referenced over the same type identifier and not their instance locator.

C. Interface Compatibility

All services use the same Application Programming Interface (API) as interface (Sec. IV-B). The use of common data access functions inherently enables composing all services (R0). This enables and facilitates (R5) the mash up of services. In addition, the use of abstract data model interfaces allows exchanging service implementations transparently.

The service compatibility goes beyond pure API compatibility. The information model that is introduced in Sec. IV bases all data nodes with values on the two data types *text* and *number* [14]. By implementing the semantics for those two types, a service can access all data available over the KA middleware. For more specific knowledge on the obtained data, specific semantics have to be included into the service logic by its developers. For supporting the semantic understanding the UIDs in the CMR can be linked to external semantics such as domain-specific ontologies [14], [31].

D. Interface Polymorphism

Service interfaces are called polymorphic if they inherit from different abstract interfaces and can be used as each of them [27]. This mechanism is also called *subtyping* [33].

A problem with subtyping is that in a classic function-based SOA a new interface can only inherit from those existing abstract interfaces that do not have function names in common. Otherwise unintended shadowing occurs in the best case. Our abstract interfaces do not have such a problem. Existing data models can simply be composed by adding a named sub node of the desired type in a new data model. As an example, a light controller could contain the data model type */lighting/light* several times under different node names in its own data model. Through the hierarchical structuring within the data models, each node spans its own namespace identified by its name.

Via the discovery mechanism described in Sec. V-B each node can be discovered as if it would be independent from the others though all interfaces are offered via the same service. The described polymorphism via hierarchical inheritance of interfaces becomes possible since abstract service interfaces are not service-centric but data-centric in our design. Each subtree of a data model can be discovered independently of the service implementing it. Which service implements a data model is hidden by the locator-id-split that is introduced with the search for type identifiers.

An interesting effect of the described possibility for subtyping is that an implementation of a service can remain compatible with previous versions of its abstract interface easily. Therefore the designer of the data model only has to include a node with the type of the old abstract interface (data model) and connect nodes to functionality. Via the data-centric approach it is transparent for a calling service if it interacts with a subtree of a newer service interface since only the locator changes

In terms of portability the described subtyping results in a significantly more powerful abstract interface compared to classic function-based interfaces. The UIDs of a description of a WS typically refer to the entire interface [9]. In our approach coupling can happen on *any* level of the data model. Interfaces automatically contain each other as they are constructed by reusing existing interfaces (Sec. V-A). If a service wants to couple with another service that offers data of a certain type it can couple with any service that contains the type on any level of its data model. In classic SOAs this would only work for the top level. This enhanced service compatibility increases the portability of services as it becomes more likely that a service finds the data it needs in subtrees of data models of different services in different smart spaces.

E. Semantic Polymorphism: Filter

The unified access to the data (Sec. IV-B) enables another kind of polymorphism that enhances the usability of SOAs following our proposed design: semantic polymorphism. The term *semantic polymorphism* describes that the same data is provided in different formats – possibly by different services.

As an example, a service A may offer a temperature in Fahrenheit. Another service B may request a temperature in Celsius. Typically service B's dependency could not be resolved as no temperature in Celsius is offered. Using the synchronous coupling of the virtual nodes (Sec. IV-B), a service C could simply be introduced that performs the translation between the two representations of a temperature. It would consume temperatures in Fahrenheit or Celsius and produce temperatures in Celsius and Fahrenheit. With this service the coupling can be realized now as service A $\rightarrow$ service C $\rightarrow$ service B. Service C acts as a filter [34] that provides the sensor data in another format (see red line in Fig. 2).

The described filtering is not only attractive for transformations between different representations of the same physical phenomenon but also for providing information on different levels of abstraction. In the energy saving scenario a service may provide information if a sensor currently detects a movement. Another service may provide information if someone is likely to be in the room. Another service may produce information if the lights can be switched off. Though being very different, all information could be derived from the readings of the same motion-detecting sensor. By introducing semantic filters that include reasoning logic [19], semantically incompatible services become compatible.

Enabling filtering facilitates the development of management functionality (R5). It significantly enhances the interface compatibility and thereby portability of service.

F. Latency

The novel coupling mode over virtual nodes (Sec. IV-B) introduces direct service coupling over a descriptive interface. This results in low latency enabling the decomposition of complex and time-critical management functionality into different modules (R4).

As all services use the identical components for communication, the expected latency of a service cascade (see Fig. 2) can be computed over a set S of services: $(|S|-1) \times t_{\text{KA-coupling}} + \sum_S t_{computation}$.

G. Simplicity, Security, Robustness

Having the management of the data model instances wrapped in a module (KA, Sec. IV-A) that can simply be used by a service developer allows focusing on the implementation logic. Freeing service developers from implementing remote access, and security facilitates not only the programming but makes services more secure and robust since common components are used (R5). Externalizing functionality makes services smaller enabling the development of better understandable, more robust code.

H. The whole picture

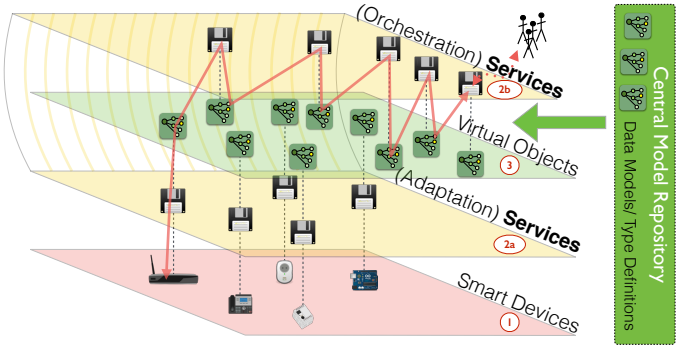

Fig. 2. A Data-Centric Service-Oriented Architecture for Managing Things.

Fig. 2 shows the layers of a classic management architecture with managed devices at the bottom (1), managing agents in the next layer (2a), and management services on top (2b). The green layer (3) is typically not explicitly shown as it is part of the services.

Our proposal is to make the data models on layer 3 the explicit abstract service interfaces. The interfaces are shared over and instantiated from the CMR on the right. As can be seen, each service has exactly one data model as abstract interface. From a service coupling perspective it does not make a difference if a service interfaces a smart device or not. Therefore the planes 2a and 2b become a single plane as illustrated with the connecting lines in the back. All services are equal in our architecture, and communicate using the same methods to access the enhanced distributed blackboard in layer 3 [27].

The red line shows a typical information flow when executing a functionality. The top right service may be the user interface of the energy saving service. It uses data that is preprocessed by different other services (Sec. V-D, Sec. V-E) and finally originates at the smart device on the bottom left.

VI. Usability Evaluation

The presented SOA architecture is fully implemented as Java peer-to-peer system [35]. The implementation was used to evaluate the usability of the proposed approach. The following user study results were collected with 8 computer science students in early 2014, and confirmed with 22 students in late 2014/ 2015. The study tasks comprise building a smart device, writing its firmware, creating the corresponding data model (Sec. IV), writing an adaptation service, and implementing management with modular services (Sec. V).

The students defined their own scenarios such as correlating the load of a PC with the room temperature and the logged in users for giving warnings over a speaker. The upper observed time bounds were: data model creation (Sec. IV) <90min, adaptation service <180min, and management services <120min. There is no evaluation of this exact use case in literature. However, the authors of [25] made a user study of apparently comparable complexity with their One.World system. Their measured overall time for the software implementation is 256h. The implementation of all described functionality from scratch using our SOA management mechanisms took each of our student teams below 6.5h overall.

The study shows that our proposed approach is feasible and provides good usability (R5). This assumption from the quantitative analysis of the teams is confirmed by a qualitative evaluation using a questionnaire. Though not having previous knowledge in the domain, all participants reported a very good usability of the proposed data-centric SOA approach.

VII. Conclusion

This paper introduced a data-centric service-oriented architecture for implementing services for managing things. The core proposal of this paper is using data models as abstract service interfaces and keeping the methods to access the data fixed. A major difference to existing NM approaches is that this paper's SOA introduces a *simple-to-use* way to mash-up complex management functionality via hierarchical cascading of management (sub-) services.

Following Perlis' epigram that "simplicity does not precede complexity, but follows it" [36], this work hopefully facilitates and enables implementing complex management tasks in current and future domains such as IoT or Pervasive Computing.

ACKNOWLEDGMENT

This research has been supported by the German Federal Ministry of Education and Research (BMBF) (reference 01IS13019G) within the EUREKA ITEA 2 project Building as a Service (BaaS) (ITEA2 No. 12011), the EU FP7 Network of Excellence in Internet Science EINS (Project No. 288021), and the German Federal Ministry of Education and Research, projects IDEM (reference 01LY1217C).

REFERENCES

[1] K. Lyytinen and Y. Yoo, "Issues and Challenges in Ubiquitous Computing ," *Communications of the ACM*, 2002.

[2] G. D. Abowd, "What next, ubicomp?: celebrating an intellectual disappearing act," in *UbiComp '12*. ACM, 2012.

[3] M.-O. Pahl and G. Carle, "Taking Smart Space Users Into the Development Loop," in *HomeSys 2013 (UbiComp 2013 Adjunct)*, Zürich, Switzerland, 2013.

[4] M. Weiser, "The Computer for the 21st Century," *Scientific American*, Sep. 1991.

[5] D. Harrington, R. Presuhn, and B. Wijnen, "An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks," RFC 3411, Internet Engineering Task Force, 2002.

[6] R. Grimm, J. Davis, and B. Hendrickson, "Systems Directions for Pervasive Computing," *Hot Topics in Operating Systems*, 2001.

[7] J. Famaey, S. Latre, J. Strassner, and F. De Turck, "A hierarchical approach to autonomic network management," *NOMS Workshops*, 2010.

[8] P. N. Johnson-Laird, *The Computer and the Mind: An Introduction to Cognitive Science*. London: Fontana Press, 1993.

[9] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications, 1st edition*, 2004.

[10] R. Fielding, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, 2002.

[11] M. Hertis and M. B. Juric, "An Empirical Analysis of Business Process Execution Language Usage," *Software Engineering, IEEE Transactions on*, vol. 40, no. 8, pp. 738–757, 2014.

[12] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Tech. Rep., 1991.

[13] J. P. Deschrevel, "The ANSA model for trading and federation," *Architecture Report APM*, 1993.

[14] M.-O. Pahl and G. Carle, "Crowdsourced Context-Modeling as Key to Future Smart Spaces," in *NOMS 2014*, 2014.

[15] M. Knappmeyer, S. L. Kiani, E. S. Reetz, N. Baker, and R. Tonjes, "Survey of Context Provisioning Middleware," *Communications Surveys & Tutorials, IEEE*, 2013.

[16] P. Bellavista, A. Corradi, M. Fanelli, and L. Foschini, "A survey of context data distribution for mobile ubiquitous systems," *Computing Surveys*, 2012.

[17] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang, "Middleware for pervasive computing: A survey," *Pervasive and Mobile Computing*, 2012.

[18] Y. Liu, J. Goncalves, D. Ferreira, S. Hosio, and V. Kostakos, "Identity crisis of ubicomp?: mapping 15 years of the field's development and paradigm change," in *UbiComp '14 Adjunct*, 2014.

[19] J. Strassner, N. Agoulmine, and E. Lehtihet, "FOCALE: A Novel Autonomic Networking Architecture," 2006.

[20] J. Strassner, J. W. K. Hong, and S. van der Meer, "The design of an Autonomic Element for managing emerging networks and services," in *ICUMT '09*, 2009.

[21] Z. Movahedi, M. Ayari, R. Langar, and G. Pujolle, "A Survey of Autonomic Network Architectures and Evaluation Criteria," *IEEE Communications Surveys & Tutorials*, 2012.

[22] L. Baresi, A. D. Ferdinando, A. Manzalini, and F. Zambonelli, "The CASCADAS Framework for Autonomic Communications," in *Autonomic Communication*, 2009.

[23] C.-F. Sørensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, and H. Duran-Limon, "A context-aware middleware for applications in mobile Ad Hoc environments," in *MPAC '04*, 2004.

[24] J. E. Bardram, "The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications," in *Pervasive Computing*, 2005.

[25] R. Grimm, "One.world: Experiences with a Pervasive Computing Architecture ," *Pervasive Computing*, 2004.

[26] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *CoNEXT09*, 2009.

[27] A. S. Tanenbaum and M. Van Steen, *Distributed Systems Principles and Paradigms*. Prentice Hall, 2002.

[28] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1985.

[29] H.-G. Hegering, S. Abeck, and B. Neumair, "Integrated Management of Networked Systems," 1999.

[30] U. Aßmann, S. Zschaler, and G. Wagner, "Ontologies, Meta-models, and the Model-Driven Paradigm," in *Ontologies for Software Engineering and Technology*. Springer, 2006.

[31] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni, "A survey of context modelling and reasoning techniques," *Pervasive and Mobile Computing*, 2010.

[32] W. Ramírez, X. Masip-Bruin, M. Yannuzzi, R. Serral-Gracia, A. Martinez, and M. S. Siddiqui, "A survey and taxonomy of ID/Locator Split Architectures," *Computer Networks*, vol. 60, pp. 13–33, 2014.

[33] L. Cardelli, "A semantics of multiple inheritance," *Semantics of data types*, 1984.

[34] L. Bergmans and M. Aksit, "Composing crosscutting concerns using composition filters," *Communications of the ACM*, 2001.

[35] "Website of the Distributed Smart Space Orchestration System," accessed 2015-02-02. [Online]. Available: http://www.ds2os.org/

[36] A. J. Perlis, "Special Feature: Epigrams on programming," *SIGPLAN Notices*, vol. 17, no. 9, Sep. 1982.