2019 International Conference on Networked Systems (NetSys)

Information-Centric IoT Middleware Overlay: VSL

Marc-Oliver Pahl, Stefan Liebald
Technische Universität München
Email: {pahl,liebald}@s2o.net.in.tum.de

***Abstract*—The heart of the Internet of Things (IoT) is data. IoT services processes data from sensors that interface their physical surroundings, and from other software such as Internet weather databases. They produce data to control physical environments via actuators, and offer data to other services.**

More recently, service-centric designs for managing the IoT have been proposed. Data-centric or name-based communication architectures complement these developments very well. Especially for edge-based or site-local installations, data-centric Internet architectures can be implemented already today, as they do not require any changes at the core.

We present the Virtual State Layer (VSL), a site-local data-centric architecture for the IoT. Special features of our solution are full separation of logic and data in IoT services, offering the data-centric VSL interface directly to developers, which significantly reduces the overall system complexity, explicit data modeling, a semantically-rich data item lookup, stream connections between services, and security-by-design. We evaluate our solution regarding usability, performance, scalability, resilience, energy efficiency, and security.

***Index Terms*—Internet of Things, middleware, data-centric, service-centric, name-based, information-centric**

I. Introduction

In the Internet Protocol (IP) based Internet, communication is host-based. Most applications in today's Internet are not host-centric but data-centric [1]. Examples are the World Wide Web (WWW), video or audio streaming. For these applications, it is not relevant from where data actually comes but to get the right data. A data-centric design fits better. [2]

For more than 10 years, researchers work on Information Centric Networking (ICN) [1]. ICN does not address hosts but data chunks. Typically, in-network caches improve latency, scalability, reliability, and energy efficiency of ICN systems compared to host-based addressing. Characteristic aspects of ICN designs are *naming*, *caching*, *decoupling of producer and consumer*, and built-in *security*.

The Internet of Things (IoT) is a part of the Internet that continuously gains importance. It can be characterized by a huge amount of globally distributed sensors and actuators that enable software services to interact with physical spaces. The IoT is a complex domain due to the distribution of its devices, and due to the heterogeneity of application scenarios [3]. Its traffic often consists of small data chunks that are exchanged between devices and services that orchestrate them [4].

Many IoT applications benefit from ICN-principles. Therefore, several ICN for IoT designs have been proposed in the past [5], [6]. ICN and the related Delay Tolerant Network (DTN) designs can enable lower latency, higher resilience, better scalability, and less energy usage [2]. All these properties are desired for the IoT. In fact, ICN principles are even more desirable for the IoT than for other Internet applications as the IoT has typically less, and less reliable resources.

978-1-7281-0568-0/19/$31.00 ©2019 IEEE

Similar to [7] in this work we apply ICN principles at the edge of the Internet, more precise at site-local IoT system level. *We present the data-centric Virtual State Layer (VSL) middleware*. It pushes ICN's decoupling of data from hosts further by even decoupling data from services on a host. We show how the VSL provides ICN/ DTN properties.

For compatibility reasons with existing IP-based IoT installations, we implemented our pilot as self-organizing Peer-to-Peer (P2P) overlay. It enables data-centric inter-service communication within and in-between IoT hosts. The VSL uses a hierarchical *naming* scheme. For consistency reasons, the VSL stores data at the source only at the moment. Work on suitable *caching* mechanisms is ongoing and discussed.

The VSL fully *decouples data producers and consumers*. It enables access via get and set, subscriptions to changes, and stream connections. Pushing the data-centric principle further, the VSL fosters a full separation of service logic and data. As consequence it can deliver service data even when a service is not running. This increases the resilience and possibly the energy efficiency of IoT nodes.

The VSL provides *security*-by-design [8], which is only possible as service logic and data are decoupled, and all inter-service communication goes through the VSL.

Major contributions of our approach are:

- Full separation of service logic and data.
- Offering the data-centric interface as Application Programming Interface (API), reducing the overall complexity significantly.
- Explicit data modeling.
- A distributed, semantically-rich data item lookup.
- Stream connections between services.
- Security-by-design.

The paper introduces the VSL system architecture (section II), including service and system model, API, data structuring, naming, placement, caching, discovery, transformation, separation of logic and state, and security mechanisms. Section III discusses the usability, performance (latency), scalability (throughput), resilience and energy efficiency, and security of our solution. Section IV discusses relevant state of the art.

II. System Architecture

We structure our architecture presentation similar to the assessments in the surveys [5], [6]. The fundamental principle behind our architecture is *modularity*. Services using our mid-

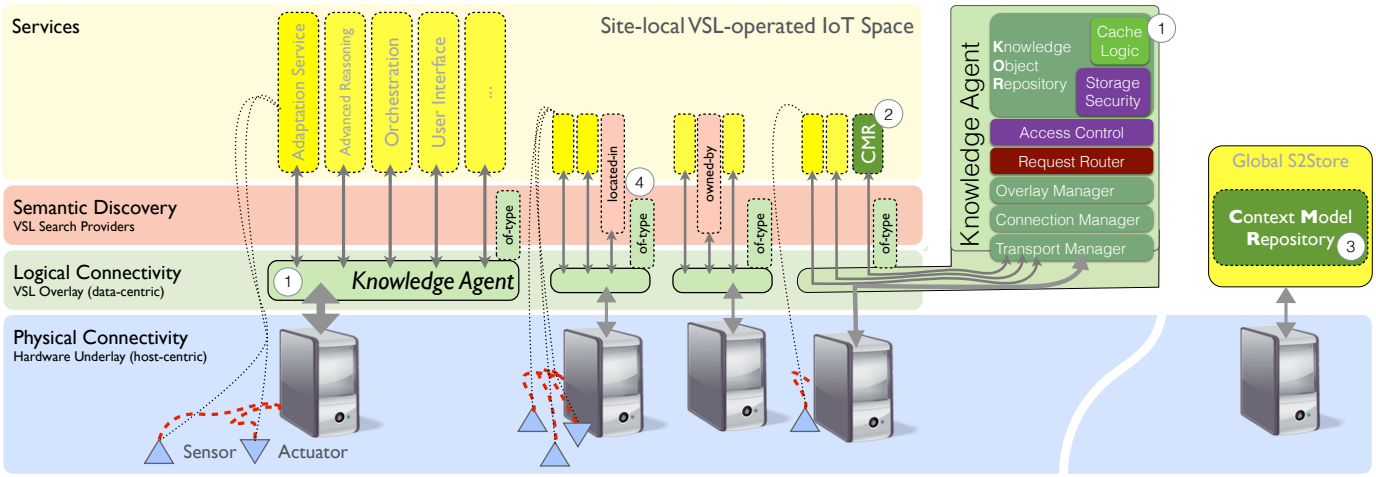

Fig. 1. Data-Centric Virtual State Layer IoT System Architecture.

dleware build complex functionality by mashing-up smaller building blocks. So does the VSL middleware itself.

A. Service Model

More recently, microservice-based approaches for managing the IoT were proposed [9], [10]. In a microservice-based IoT, several microservices federate dynamically to implement complex applications. We assume that the available resources of IoT devices will continuously increase as we have seen it for Mobile Computing smartphones [11]. IoT nodes such as light switches will soon have enough computational power to run IoT microservices.

To manage the complexity of the IoT we target a microservice architecture where independently running IoT services dynamically federate to implement IoT scenarios [9], [10], [12]. Relevant service types include:

- *Adaptation Services* interface IoT hardware and software components, e.g. for obtaining Internet weather data,
- *Reasoning Services* filter and enrich existing data,
- *Orchestration Services* federate and compose services,
- *User Interface Services* provide user controls.

Following, we illustrate the concept with a climate control. An Orchestration Service for temperature control federates an Adaptation Service with a thermometer, an Adaptation Service that actuates the valve of a heater, and possibly one or multiple Reasoning Services for transforming units. A User Interface Service can be involved to get the humans in the control loop.

In our setting all services are autonomous, enabling them to co-exist independently. Data producers and consumers are fully independent. In a time-sharing manner they federate, making a shared use of the installed hardware for implementing different applications possible.

For such a setting, a suitable communication infrastructure needs to provide a dynamic discovery mechanism that is driven by the local IoT applications [13]. It requires providing suitable inter-service coupling mechanisms that do not limit the applications [9], [14], and providing a suitably high security level regarding storage and exchange of data [15]–[18].

B. System Model

Figure 1 shows our system architecture. It consists of four layers. The lowest layer (blue) contains the IoT hardware: IoT compute nodes (shown as computers), sensors, and actuators (shown as triangles). This layer provides the *Physical Connectivity* between the participating nodes.

The next layer provides *Logical Connectivity*. It implements our data-centric VSL that spans distributed IoT nodes via its so-called *Knowledge Agents* (KA) (1). The KAs offer a data-centric interface to services. This API enables querying data items from the local Knowledge Object Repository (KOR), and via the connected KAs from their KORs that store the IoT data (section II-F).

For being compatible with existing deployments, we implement the KA connectivity as Peer-to-Peer (P2P) overlay. It uses IP multicast and unicast connections for exchanging information. The VSL uses IP multicast to maintain the overlay and unicast for directed data exchanges.

The right side of fig. 1 details the components of a KA (1). The transport manager uses protocols such as HTTP over TCP/IP as transport. The *Transport Manager*, *Connection Manager*, and the *Overlay Manager* maintain the P2P overlay. The entire inter-node connectivity is encapsulated in these modules.

Our transport can easily be exchanged with other communication protocols. It is simply our way to implement a communication channel between distributed nodes. With larger multi-hop installations, replacing the IP layer with a name-based routing approach could become interesting in the future. The architecture is prepared for that.

From now on we assume that the KAs can communicate over an abstract interface that allows data exchange. As northbound interface to services and as southbound interface to other KAs, the VSL overlay provides data access based on unique hierarchical data item identifiers.

The next layer offers *Semantic Discovery*. Services communicate by accessing each other's data items. However, for discovery they use a semantic lookup based on predicates

such as function identifier, data type, location, or owner of a data item. Following a modular approach, such discoveries are implemented as so-called *Search Providers* that can be plugged in at runtime (section II-H).

The top layer contains the services. Services only connect to the next, typically locally running KA to get access to the data-centric VSL communication primitives (section II-C).

We do not target replacing protocols towards resource constraint devices such as 802.15.4, KNX, or CoAP. Instead, we interface them with *Adaptation Services*. They run on resourceful nodes that have corresponding hardware interfaces as shown on three nodes in the Physical Connectivity layer.

We overcome IoT device protocol heterogeneity by introducing digital twins for each managed entity including IoT software and hardware (section II-D). A digital twin is a VSL data structure exposing all properties that can be monitored and controlled. Combined, the VSL data represent all capabilities and states of an IoT space. Therefore, we call our abstraction *Virtual State Layer (VSL)*.

Services manage the consistency of the digital twins in the VSL with their managed counterparts. These adapter services have a device-specific interface at their southbound interface, and a standardized VSL context model, which describes the digital twin, as northbound interface (section II-D). They reflect changes between the context model instance and the entity at the southbound interface that could be a smart power plug, or an email interface.

C. VSL API

The VSL implements a data-centric blackboard communication pattern. Services communicate over data items that are managed by the VSL KAs, and accessed through its API.

IoT scenarios are only limited by the creativity of their developers and the available hardware. For enabling the implementation of diverse applications, the VSL offers three inter-service couplings over its data nodes:

1) *Asynchronous Communication via Get/ Set* enables services to store and retrieve data items fully decoupled in time and space. The persistent storage of the VSL overcomes the time aspect, the overlay's location transparency the space boundary.
2) *Synchronous Communication via Publish/ Subscribe* allows services to subscribe on changes of given data nodes. Once the current value of the node changes, a callback into the subscribing service is triggered.
3) *Synchronous Communication over Sockets* enables services not only to communicate synchronously but emulates function calls with parameters over so-called *Virtual Nodes* [14].

The last mechanism is very powerful and typically not found in data-centric middleware. It is the base for the modular design of the VSL. The data discovery *Search Providers* (section II-H) make heavy use of this feature. In short, a function call is implemented as access to a virtual VSL address: `vsl://ka1/service2/function/param1/param2/`

D. Data Structuring

VSL data represents the digital twins of the entities a service manages. In a data-centric design, data becomes the interface of a service (section II-C) [14].

For enabling the intended dynamic mashup of services, a convergence of the VSL data models e.g. for lamps is required. Having one data type per semantic functionality is essential to decrease the software development complexity. As an example, if all services that control lamps offer a different data model as semantic interface, an application developer who wants to shut off all lamps has to interface all different models. In [19] we discuss methods to solve this.

A methodology for structuring VSL data is not only required for interface convergence but also for data discovery [13]. The VSL offers an object-oriented information model that facilitates the creation of service data models [19]. Coming from the application development perspective, these data models are called *VSL Context Models*.

A *Context Model* is a hierarchical data structure. The VSL information model uses only the two predicates *is-a* and *has-a*. Based on the concepts *text*, *number*, *list*, and *composed*, complex data types can be built. Such types can be named, acting as functionality tags, e.g. `/lamp/dimmableLamp`.

The data type *composed* implements the *has-a* relationship by aggregating already defined types as subnodes of its tree root. The data type *list* is the only data structure that can be altered at runtime within potential restrictions.

All other node types implement *is-a* relationships. They can be derived for introducing semantic identifiers, e.g. "isOn" for a binary value defined as $\{0, 1\}$ restricted set of type number.

For enabling reuse, portability, and standardization, VSL context models are shared over a global directory that is called Context Model Repository (CMR) [19]. Its purpose is to standardize model definitions by acting like yellow pages for context models. Each model therefore has a unique *ModelID* and a corresponding specification in the CMR. See fig. 1 on the right (3). Developers can look existing models up and reuse them in their own creations.

Listing 1 shows the definitions of the VSL Context Model types `lamp` and `dimmableLamp`. The listing illustrates how the inheritance facilitates the definition of the `dimmableLamp` type. It also shows how inheritance can be used to stay compatible with existing types [19].

Via subtyping the `dimmableLamp` model remains compatible with the `lamp` model. As the data models are the interfaces of the VSL services, this compatibility means that a service that can control `lamps` can still control `dimmableLamps`. In other words, `dimmableLamp` implements the semantic `lamp` interface.

```
1  <lamp type="basic/composed">
2    <isOn type="isOn">0</isOn>
3  </lamp>
4
5  <dimmableLamp type="lamp">
6    <dimValue type="dimValue">100</dimValue>
7  </dimmableLamp>
```

Listing 1. Definition of two VSL context models.

```
<service42 type="dimmableLamp, basic/composed">
  <isOn type="isOn, basic/number" restriction="
      maxValue=1, minValue=0">1</isOn>
  <dimValue type="dimValue, basic/number">50</
      dimValue> </service42>
</service42>
```

Listing 2. VSL instantiation of the dimmableLamp model.

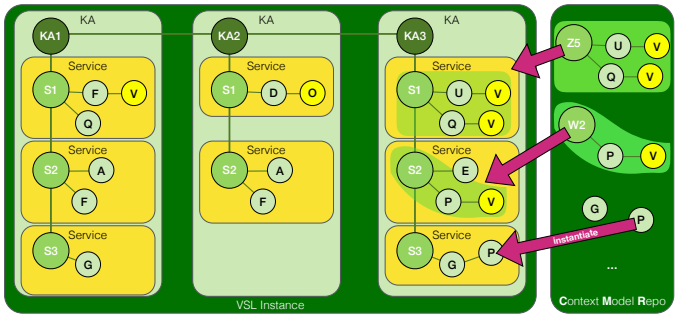

Fig. 2. Context Data managed by distributed KA peers and the CMR.

Listing 2 shows an instance of the *dimmableLamp* VSL Context Model within an IoT site. Figure 2 illustrates how data that is detailed in listing 2 is distributed over different KAs. The figure shows three IoT computing nodes running a VSL KA each. KA1 and KA3 run three services S1-3 each. KA2 runs two services. The context models are specified in the CMR as is indicated on the right. For each VSL modelID there is a tree of data nodes, e.g. `Z5`.

Each VSL service is associated with one specific context modelID. When a service gets started, the corresponding context model is loaded via the local CMR from the global one as shown with the arrows. The service's KA then creates a local instance that becomes the service's state space. From that point in time the CMR is not needed anymore.

To increase the performance of the instantiation, context models are cached site-locally. This is shown in fig. 1 with the site-local CMR node (2). It is the caching interface for all local KAs to the CMR.

E. Naming

The VSL uses a *hierarchical namespace*. VSL addresses identify the original source of a data item. Following the architecture shown in fig. 1, the VSL names identify data nodes belonging to a service on a KA. Though the VSL manages data locally, it is possible to connect worldwide distributed VSL IoT Spaces. Therefore, we add the `siteID` for getting globally unique identifiers per data item: `vsl://[siteID]/[kaID]/[serviceID]/[subNodeAddress]/...`

The *siteID* is unique for each edge-based VSL site. The *kaID* is unique for each Knowledge Agent. The *serviceID* is node-locally unique for each service. The *subNodeAddress* is unique within the service's data.

The hierarchical names facilitate debugging. Though they seem to bind data to KAs, the names do not necessarily locate VSL data model instances (section II-D). In fact, meaningless hashes could be used as address for a model's root node.

F. Data Placement

The VSL uses a source placement, storing data always at the source. Fostering microservice mashups makes inter-service communication highly frequent. Therefore having data close to the processing services is beneficial. The logical consequence of the VSL's separation of service logic and data (section II-J) is storing data at the KA where a service runs.

Locality is especially beneficial to support low latency and high throughput via the IoT's heterogeneous communication links. Often a KA runs locally on the same IoT node like a service. However, with the VSL's full location transparency, data can be stored on any KA.

G. Data Caching

Each VSL data item has a unique identifier (section II-E). When a service provides new data items, e.g. values for a given address, they get unique IDs. We implement this by attaching a version number to the VSL node address, and pointing the address without trailing number to the newest value. Consequently, VSL data can easily be cached.

With its subscribe model (section II-C), the VSL offers a suitable mechanism for keeping on-path caches up-to-date. However, this comes at communication and processing costs.

It is difficult to guarantee that a cached value is the newest available value. In addition, cache coherency on write is a challenge. This case happens when a value of an actuator, e.g. `door/isOpen` is set.

For consistency reasons, and as we only have very limited multi-hop scenarios site-locally, we do not cache data in the VSL at the moment. However, we currently experiment with different caching strategies for decreasing the latency and increasing the resilience in case of KA failures [20].

H. Data Discovery

The IoT is a dynamic environment. Entities join and leave frequently. In addition, our microservice based IoT orchestration requires services to discover each other frequently in order to run scenarios on the shared infrastructure (section II-A).

For meeting the described dynamics, VSL services do not bind statically using the VSL addresses but run a lookup to identify suitable currently available coupling candidates. All services communicate via the VSL API (section II-C) and use the standardized context models (section II-D). This provides interface compatibility that enables mashups.

Built-in, the VSL uses type tags for data nodes. Each data node can have one to many data types [19]. As briefly introduced in section II-D, a VSL node typically has a data type identifier (e.g. `basic/number`, and a functional identifier (e.g. `lightin/hue123/dimValue`).

A type-based example lookup for the VSL type *lamp* is `get /search/type/lamp`. As result, the requester gets all addresses of type lamp, e.g. the instance address `/SID123/KA4/service42`. See listing 2.

A discovery on these tags is provided by the KAs. However, the diversity of IoT applications often requires different or additional discovery predicates than the described `of-type`.

The VSL offers a plug-in concept for so-called *Search Providers* [13]. A search provider implements a predicate such as `located-in`, or `owned-by`.

Via Virtual Nodes (section II-C), Search Providers can be plugged-in transparently at runtime. Discovery requests are automatically routed to the right Search Provider by the KAs. This request routing becomes possible as the discovery of Search Providers uses the built-in type search, e.g. `system/searchProvider/locatedIn`.

All Search Providers offer the same interface (section II-D). Therefore, we can offer a first order logic *search federation service* that enables *complex semantic discoveries* for VSL data node identifiers (section II-E). A sample query is `get/search/metaQuery/locatedIn/livingRoom/AND/ownedBy/mop/AND/of-type/light`. The first parameter identifies the search provider, requests should be routed to. The other parameters are the search objects [13].

Type searches are most frequent. They enable decoupling services from their running location, which enables the desired late coupling. For a fast type search, the VSL periodically replicates the information of all available data nodes and their types between all KAs in a site. This enables executing the type search locally on a KA guaranteeing a fast result. In contrast to a distributed search, e.g. via flooding, the local discovery is also terminating deterministically. With flooding it is uncertain, if and when all nodes responded.

For non-built-in Search Providers it is up to the developer to implement a suitable replication and load balancing. The built-in type search has full #KA-resilience. All discovery queries return a list of VSL identifiers (section II-E). The addresses are then accessed by a service in order to get the desired data.

I. Data Transformation

The VSL's digital twins (section II-D) bridge a large part of the IoT heterogeneity already. However, on a higher semantic level such functionality is highly application specific. An example is a service that needs a temperature in Celsius or in Fahrenheit. Another service requires to know if it is a nice day, but locally only a brightness sensor, a temperature sensor, a rain sensor, and a wind sensor are available.

With the *VSL Reasoning Services* (section II-A) a category of services exists that does such data transformations. In combination with the VSL Data Discovery (section II-H) such data transformation can be automatically included in the service mashup process. When a suitable *Reasoning Service* is available, it will be discovered and invoked, and consecutively transparently transform the available data as requested [14].

J. Separating Logic and State

The VSL offers services the functionality to store and retrieve data items from all over the network. Pushing the data-centric principle further, we foster to fully separate service logic and state. All data is then managed *within* the VSL.

This makes the VSL architecture resilient against service failures on nodes. It also enables potentially saving energy by stopping services that are not required at the moment. Their data remains available. Most important, it takes the burden to implement adequate service access security from the developers. The VSL manages all communication and can therefore enforce security by design [15]–[17].

K. Security

Different from most data-centric IoT designs [5], [6] the VSL consists of data managing agents, the KAs. These agents self-organize in a P2P manner. IoT data is inherently privacy-critical, as it typically consists of measured or otherwise collected personal data, e.g. temperature and music preferences. By outsourcing data and communication security into the VSL (section II-J), a crowdsourced development of IoT services/ Apps [12], [21] becomes realistic.

To implement the desired and required security by design [8], the VSL uses certificates that are pinned to all components: each service, and each KA. The distribution and renewal of these certificates is automated [15], [17].

The used X.509v3 certificates enable authenticating the software components. They also enable establishing TLS-secured communication between the KAs (section II-B). Finally, they enable the secure exchange of keys for encrypted stores, e.g. encrypted local databases that store the VSL data.

On top of this basic security, the VSL implements group-based access control for read and write accesses to VSL data nodes. Each service has a pre-defined set of group identifiers that are matched with the identifiers in accessed data model instances [15]. As the KAs are trusted, they can effectively mediate all accesses [16].

Not only the type information but also the access modifiers are synchronized between the KAs. Consequently, VSL node discoveries filter the results based on a service's access IDs at the source already. As a result, only those VSL addresses (section II-E) are returned that are accessible by the service.

III. Evaluation

The VSL is fully implemented in Java. This enables running KAs on diverse operating systems. It also facilitates the development of functionality. The cost is a larger executable footprint, and a lower performance.

For the quantitative measurements we used a testbed with "unlimited" resources: i5 computers with SSD, 4 Gigabyte RAM, and 1GBps Ethernet. Since future IoT device generations will be faster we want to prevent bias from scarce resources.

A. Usability: Continuous Evaluation

The VSL targets facilitating the implementation of IoT scenarios. We evaluated this with more than 150 student testers that were beginners with the IoT, somehow familiar with Java, and did not use the VSL before.

To illustrate the use of the VSL API, listing 3 shows a data retrieval. The *VslConnector* connects the service to its KA. Its required setup code is below 10 lines and not shown. In line 2 we search the VSL for all data items of type lamp using the special `/search/type` address prefix with the searched

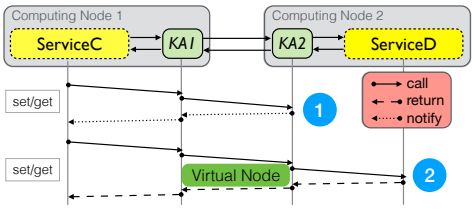

Fig. 3. Measurement Setup.

Operation	local		remote	
	regular	virtual	regular	virtual
get	1.3 ms	1.6 ms	10.4 ms	10.8 ms
set	1.9 ms	2.6 ms	9.3 ms	10.0 ms

Table I
Average delay of local/remote get/set requests (20000 each).

type as suffix. In line 4-6 for each retrieved address we print whether the lamp is active or not.

Typical VSL service modules developed by the students were below 100 lines of code. With our powerful data-centric abstraction that suffices to implement microservices and their mashups for complex IoT scenarios.

```
1 public void printLamps(VslConnector con) {
2   String[] addr = con.get("/search/type/lamp")
        .getValue().split("//");
3   //e.g. ["/ka1/lamp1","/ka2/lamp2",...]
4   for(int i=0;i<addr.length;i++){
5     System.out.println(
6       con.get(addr[i]+"/isOn").getValue()); }}
```

Listing 3. Example VSL call to print the state of all available lamps.

The students get a tutorial and have to implement a complex use case on their own. All manage to do so in less than 20h. This is remarkable as in the only quantitative study with a comparable middleware, experienced developers implemented a workflow of similar or lower complexity only in 120h [22].

73% rated the VSL API as well suitable or even easy-to-use for beginners. 67% described the task difficulty as well doable or easy.

Both results confirm that the VSL is a powerful abstraction that is so simple that beginners can start using it within short time. At the same time it shows how powerful the programming abstraction is as it also fits well for the individual extensions made by the students.

B. Performance

The time-sequence diagram in fig. 3 illustrates the VSL communication via regular node accesses (1) and Virtual Node accesses (2). For both we measured 20000 times the latency for local and remote accesses. Figure 3 shows the remote case. In the local case, both KAs are merged.

For get operations we measure the time between issuing the command in the client and its return. For the *set* operation we wait until the operation finishes updating the value without returning an error. Table I shows the measured average delays.

The evaluation shows that the performance for requests on target services running on the same node is around 1.3-2.6 ms. For remote requests we achieve delays around 10 ms, which is quite low and suitable for IoT applications. While local get requests are slightly faster on average than set requests, it is the opposite for remote requests.

The additional overhead comes most likely from serializing the response and sending it over the network. For set requests there is only a short status code sent back in case no error occurs, with no additional payload. Subscriptions return with neglectable delay in the regular node case, making this mechanism suitable for synchronous coupling.

C. Scalability

We measure the scalability with 48 IoT services, equally distributed over 6 KAs. On each KA seven measurement clients and one accessed target run. All measurement services randomly query either the local or a remote target service.

Each client measures the throughput of 1500 get and set request bursts for regular and virtual nodes. The experiment is repeated with five, three and one clients per KA. In total around 1.2 million data points were collected. A regular IoT load can be expected to be significantly lower. The results can therefore be interpreted as stress test.

Figure 4 shows the results for node-local accesses. The values for the remote accesses are about 10 times less requests per second and 10 times longer runtime.

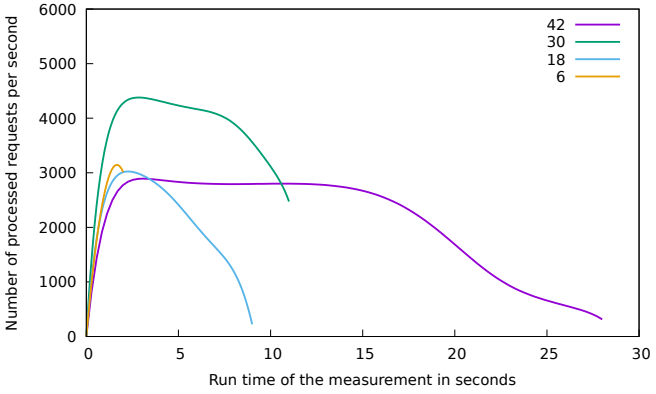

Fig. 4. Aggregated throughput of local set requests per second for 6, 18, 30, 42 concurrently running services.

D. Resilience and Energy Efficiency

Separating service logic and state (section II-J) makes data available even for offline services. The VSL's immutability of data items, and the naming scheme (section II-E) facilitate caching. We are currently looking at the effects of caching data items at the destination KAs (section II-G).

Service external storage and distributed caching enhance the availability of data, resulting in less data transfers. This increases the resilience, performance and scalability. In addition, it can reduce the energy consumption.

E. Security

The separation of service logic and service state (section II-J) enables implementing security fully within the VSL (section II-K). The result is security-by-design over the entire data life-cycle from the producer to the consumer [8], [15].

IV. Related Work

The VSL is an inter-service communication overlay that provides full data management. It offers services a data-centric abstraction using get/ set, publish/ subscribe, and stream connections. For service developers it behaves like an Information Centric Network (ICN). Therefore, ICN principles are highly relevant for us. The authors of [23] give an overview on properties of different ICN proposals.

The ICN and VSL *system models* consist of distributed nodes that route requests and can cache data. Most ICN proposals target a clean slate architecture [1], [24], [25]. Some co-existence to IP via overlays [26], [27]. For retrofitting to existing infrastructures, we consider the overlay approach more promising. Still we fully decouple the service we offer from the used IP substrate, also supporting clean slates.

Service model: ICN data caching often assumes static producer-consumer locations. There, on-path caching is suitable [1], [24]. Others use context clusters for data distribution [28], [29]. Dynamically spawning and moving IoT services result in more complex communication patterns [16].

ICN *API* access happens via publish-subscribe [1], [25], [28] and get/ set [26], [27]. In addition, we offer Virtual Nodes. The ICN *data structuring* is typically key-value [24]. Semantically richer proposals use fixed data object structures [26]. The authors of [25] propose using the Resource Description Framework (RDF). We use key value pairs with some fixed attributes like access modifiers, time stamps, and version numbers. Via our type-based search and its dynamic extensibility we achieve RDF-comparable expressiveness at significantly reduced complexity [19]. *Versioning*: For representing data items that change over time, [25] propose links that point to the newest data version. We also use this scheme.

Naming: ICNs typically use flat name spaces [1], [24], [26]. We use flat naming for our source KAs. Within a KA's name space we use hierarchical names for representing our structured data. *Routing*: ICN addresses data not hosts. Locating data happens on-path, using strategies like request flooding [1], [24], or via off-path lookups [1], [25], [26], [30]. We use our KA-local P2P overlay/ underlay address resolution. *Caching*: ICNs typically cache on-path [1], [23], [24], [26], [30]. Some solutions also enable off-path caching [25], [26]. For IoT communication relationships there is no clearly winning strategy. It depends on the installed services and conditions in a space. However, we plan to add both strategies to the KAs for increasing resilience [20].

ICN *Name Discovery* typically uses external directories [1], [25] or system internal attributes [26], [27]. The VSL's dynamically extensible semantic lookup is more powerful.

[27] propose *in-network processing* of routing-related functionality. [31] propose simple functionality such as aggregating data. Our modular data-centric plug-in concept enables more flexible on-path processing.

ICN typically provides *security* using cryptographic hashes as name parts [24], [29], validating data sources and data integrity [1], [24]–[26]. We implement security by establishing the VSL as trusted fully encapsulated system.

[32], [33] use plain ICN for IoT device communication. This is not our focus as we are in the management domain not in the control or data plane.

[3], [5], [6] assess ICN-based designs and their possible uses in the IoT. [34] discuss challenges of creating data-centric middleware for the IoT. All confirm our identified challenges, and the fit of our approach. Our architecture is different from the surveyed ones. It solves identified limitations of ICN, including semantically rich discovery, and scalability. [5], [6], [35], [36] survey solutions using data-centric principles for middleware that like us is implemented as IP overlay. They are similar to the VSL regarding the overlay aspect, and the access primitives. They mainly differ in the data modeling and discovery, caching, and security.

As a representative example, C-Dax [37] is a data-centric overlay for smart grid data exchange. Similar to us are: the inter-service communication primitives, the implementation as overly, using distributed, possibly replicated databases, and authenticating entities and encrypting communication with web technologies. Different to us is: limited data modeling and discovery, no separation of service logic and data, more complex usability for developers.

Several more-widely deployed solutions partly provide data-centric features. Data Distribution Service (DDS) [38] is an IP-based data-centric publish-subscribe architecture for machine-to-machine (M2M) communication. The VSL is closer to ICN. Through its separation of service logic and data it offers more functionality by-design such as security.

Similar to DDS, OPC UA [39], [40] implements a Service-Oriented Architecture (SOA). Services discovery uses predefined attributes. Our discovery is richer than UPC UA and DDS. Like DDS, OPC UA lacks data management.

The Constrained Application Protocol (CoAP) [41] is lightweight HTTP for resource constrained devices as in the IoT [42]. On top of its request/response-based messaging model, CoAP supports data caching and resource discovery. Discovery happens via multicast queries for device self-descriptions. The VSL information model, and the semantic discovery are much richer. The data management capabilities of the VSL go beyond those of CoAP.

MQTT [43] is a lightweight publish subscribe protocol [44]. Clients can subscribe topics at information brokers. Topics are hierarchically structured, similar to the typing approach of the VSL. Further semantic descriptions and most important, data management functionality are not part of MQTT. Security is also in the domain of the service developers.

V. Conclusion

This paper presented the Virtual State Layer (VSL) IoT middleware. It combines data-centric principles with agents for managing data on behalf of services. It uses a peer-to-peer approach for request routing to enable a retrofitting to existing infrastructures. Through its full encapsulation, using ICN routing would also be possible.

We propose a full separation of service logic and data (sections II-A, II-B and II-J), offering the data-centric interface

as Application Programming Interface for reducing the complexity (sections II-A to II-C, II-E to II-G and II-I), explicit data modeling (section II-D), a distributed, semantically-rich data item lookup (section II-H), stream connections between services (section II-C), and security-by-design (section II-K).

Our evaluation shows a high usability, low latency, good scalability, good resilience and possibly increased energy efficiency, and security as core principle (section III). We hope that our work pushes the adoption of the IoT further into the real world while providing necessary basics such as security.

REFERENCES

[1] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, p. 181, 2007.

[2] D. Trossen, A. Sathiaseelan, and J. Ott, "Towards an Information Centric Network Architecture for Universal Internet Access," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, pp. 44–49, 2016.

[3] C. Fang, H. Yao, Z. Wang, W. Wu, X. Jin, and F. R. Yu, "A survey of mobile information-centric networking: Research issues and challenges," *IEEE Comm. Surveys and Tutorials*, vol. 20, no. 3, pp. 2353–2371, 2018.

[4] M.-O. Pahl and G. Carle, "The Missing Layer - Virtualizing Smart Spaces," in *10th IEEE International Workshop on Managing Ubiquitous Communications and Services 2013 (MUCS 2013, PerCom 2013 adjunct)*, San Diego, USA, 2013, pp. 139–144.

[5] S. Arshad, M. A. Azam, M. H. Rehmani, and J. Loo, "Recent Advances in Information-Centric Networking based Internet of Things," *IEEE COMM. SURVEYS & TUTORIALS*, vol. 14, no. 8, pp. 1–34, 2018.

[6] M. Amadeo, C. Campolo, J. Quevedo, D. M. Corujo, A. Iera, R. L. Aguiar, and A. V. Vasilakos, "Information-Centric Networking for the Internet of Things: Challenges and Opportunities," *IEEE Network Magazine*, no. April, pp. 92–100, 2016.

[7] E. Borgia, R. Bruno, M. Conti, D. Mascitti, and A. Passarella, "Mobile edge clouds for Information-Centric IoT services," in *IEEE Symposium on Computers and Comm.*, vol. 2016-Augus, 2016, pp. 422–428.

[8] A. Cavoukian, "*Privacy by Design*: Leadership, Methods, and Results." *European Data Protection*, pp. 175–202, 2013.

[9] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed Smart Space Orchestration," in *Network Operations and Management Symposium 2016 (NOMS 2016) - Dissertation Digest*, 2016.

[10] D. Lu, D. Huang, A. Walenstein, and D. Medhi, "A Secure Microservice Framework for IoT," in *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE*. IEEE, 2017, pp. 9–18.

[11] R. Want, "The Power of Smartphones," *Pervasive Computing, IEEE*, vol. 13, no. 3, pp. 76–79, 2014.

[12] M.-O. Pahl, "Multi-tenant iot service management towards an iot app economy," in *HotNSM workshop at the International Symposium on Integrated Network Management (IM)*, Washington DC, Apr. 2019.

[13] M.-O. Pahl and S. Liebald, "A modular distributed iot service discovery," in *IM 2019 ()*, Washington DC, USA, apr 2019.

[14] M.-O. Pahl, "Data-Centric Service-Oriented Management of Things," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, Ottawa, Canada, May 2015, pp. 484–490.

[15] M.-O. Pahl and L. Donini, "Giving IoT Edge Services an Identity and Changeable Attributes," in *International Symposium on Integrated Network Management (IM)*, Washington DC, USA, apr 2019.

[16] M.-O. Pahl and F.-X. Aubet, "All Eyes on You: Distributed Multi-Dimensional IoT Microservice Anomaly Detection," in *2018 14th International Conference on Network and Service Management (CNSM) (CNSM 2018)*, Rome, Italy, Nov. 2018.

[17] M.-O. Pahl and L. Donini, "Securing IoT Microservices with Certificates," in *Network Operations & Mgmt. Symposium (NOMS)*, Apr. 2018.

[18] M.-O. Pahl, F.-X. Aubet, and S. Liebald, "Graph-Based IoT Microservice Security," in *Network Operations and Management Symposium (NOMS)*, Apr. 2018.

[19] M.-O. Pahl and G. Carle, "Crowdsourced Context-Modeling as Key to Future Smart Spaces," in *Network Operations and Management Symposium 2014 (NOMS 2014)*, May 2014, pp. 1–8.

[20] M.-O. Pahl, S. Liebald, and L. Wüstrich, "Machine-learning based IoT Data Caching," in *Integrated Network Mgmt. (IM), 2019 HotNSM at IFIP/IEEE International Symposium*, Washington, USA, Apr. 2019.

[21] M.-O. Pahl and G. Carle, "Taking Smart Space Users into the Development Loop: An Architecture for Community Based Software Development for Smart Spaces," in *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*. New York, NY, USA: ACM, 2013, pp. 793–800.

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

[23] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A Survey of information-centric networking research," pp. 1024–1049, 2014.

[24] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 1–12.

[25] B. Ahlgren, V. Vercellone, M. D'Ambrosio, M. Marchisio, I. Marsh, C. Dannewitz, B. Ohlman, K. Pentikousis, O. Strandberg, and R. Rembarz, "Design considerations for a network of information," *Proceedings of CONEXT '08*, no. January, pp. 1–6, 2008.

[26] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl, "Network of information (NetInf) - An information-centric networking architecture," *Computer Comm.*, vol. 36, pp. 721–735, 2013.

[27] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani, "MobilityFirst : A Robust and Trustworthy Mobility- Centric Architecture for the Future Internet," in *ACM SIGMobile Mobile Computing and Communication Review (MC2R)*, 2012, pp. 1–12.

[28] D. Trossen and G. Parisis, "Designing and realizing an information-centric internet," *IEEE Comm. Mag.*, vol. 50, no. 7, pp. 60–67, 2012.

[29] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," *ACM SIGOPS Operating Systems Review*, vol. 34, no. 2, pp. 19–20, 2000. [Online]. Available: http://portal.acm.org/citation.cfm?doid=346152.346183

[30] M. D'Ambrosio, C. Dannewitz, H. Karl, and V. Vercellone, "MDHT," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking - ICN '11*, 2011, p. 7.

[31] O. Ascigil, S. Reñé, G. Xylomenos, I. Psaras, and G. Pavlou, "A keyword-based ICN-IoT platform," in *Proceedings of the 4th ACM Conf. on Information-Centric Networking - ICN '17*, 2017, pp. 22–28.

[32] J. Quevedo, D. Corujo, and R. Aguiar, "A case for ICN usage in IoT environments," *2014 IEEE Global Communications Conference, GLOBECOM 2014*, no. September 2017, pp. 2770–2775, 2014.

[33] E. Baccelli, C. Mehlis, O. Hahm, T. C. Schmidt, and M. Wählisch, "Information Centric Networking in the IoT: Experiments with NDN in the Wild," in *1st ACM Conf. on Information-Centric Networking*, 2014.

[34] A. Lindgren, F. B. Abdesslem, B. Ahlgren, O. Schelén, and A. M. Malik, "Design choices for the IoT in Information-Centric Networks," *2016 13th IEEE Annual Consumer Communications and Networking Conference, CCNC 2016*, pp. 882–888, 2016.

[35] S. Chatterjee, "A Survey of Internet of Things (IoT) over Information Centric Network (ICN)," no. August, pp. 0–18, 2018.

[36] V. Raychoudhury, J. Cao, M. Kumar, and D. Zhang, "Middleware for pervasive computing: A survey," *Perv. and Mob. Computing*, Sep. 2012.

[37] M. Hoefling, F. Heimgaertner, M. Menth, K. V. Katsaros, P. Romano, L. Zanni, and G. Kamel, "Enabling resilient smart grid communication over the information-centric C-DAX middleware," in *Proceedings - International Conference on Networked Systems, NetSys 2015*, 2015.

[38] Object Management Group (OMG), "Data Distribution Service (DDS) Version 1.4," 2015.

[39] F. Pauker, T. Frühwirth, B. Kittl, and W. Kastner, "A Systematic Approach to OPC UA Information Model Design," *Procedia CIRP*, vol. 57, pp. 321–326, 2016.

[40] S.-H. Leitner and W. Mahnke, "Opc ua - service-oriented architecture for industrial applications." *Softwaretechnik-Trends*, vol. 26, no. 4, 2006.

[41] C. B. Z. Shelby, K. Hartke, "The Constrained Application Protocol (CoAP)," Internet Requests for Comments, RFC Editor, RFC 7252, June 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7252

[42] C. Gündoğan, P. Kietzmann, M. Lenders, H. Petersen, T. C. Schmidt, and M. Wählisch, "NDN, CoAP, and MQTT: A Comparative Measurement Study in the IoT," vol. 13, 2018.

[43] "mqtt," https://mqtt.org, undated, [Online; accessed 18-May-2018].

[44] Oasis, "MQTT Version 3.1.1," OASIS - Advancement of Structured Information Standards, Standard, dec 2015.