A Solid Architecture for Machine Data Exchange with Access Control

Justus Fries[1,*], Michael Freund[1,2] and Andreas Harth[1,2]

[1]*Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany*
[2]*Friedrich-Alexander University Erlangen-Nürnberg, Nürnberg, Germany*

Abstract
We present an architecture based on Social Linked Data (Solid) and W3C Web of Things (WoT) that enables controlled access to composite IoT devices and their data. The architecture allows for stakeholders to interact with the composite device and to read and write data, provided they are authorized. For fine-grained access control, data is stored in nested Linked Data Platform (LDP) containers that reflect the structure of the composition itself. The composition structure is derived from a hierarchy of WoT Thing Descriptions (TDs). The architecture relies on an orchestration component to couple IoT devices and the Solid data store and is straightforward to implement with existing software.

Keywords
Web of Things, Social Linked Data, Decentralized Data Exchange.

1. Introduction

The Internet of Things (IoT) is a key technology for industrial automation. IoT devices can be simple standalone devices, such as individual sensors and actuators, or more complex composite devices, such as a conveyor belt composed of motors and sensors. Individual IoT devices and the subsystems that form a composite produce dynamic data at runtime and have associated static data, such as datasheets, maintenance or usage instructions, but are constrained by their limited processing power, storage capacity and power supply.

As a result, data of standalone and composite IoT devices cannot be made available to stakeholders on the devices themselves. Data must be offloaded to other components of an IoT system. These components must provide fine-grained access control to manage the data, given the involvement of multiple stakeholders with varying levels of authority and responsibility. For example, an external maintenance worker should not have access to all data, unlike the owner of a machine. The system also needs to be able to control and orchestrate composites and their standalone IoT devices to perform tasks.

The benefit of long-term storage of data generated by and associated with an IoT device combined with fine-grained access control is that smart factories can continuously analyze and

SWoCoT'23: 1st International Workshop on Semantic Web on Constrained Things, 28th May 2023, Hersonissos, Greece, co-located with 20th Extended Semantic Web Conference (ESWC 2023)

*Corresponding author.
justus.fries@iis.fraunhofer.de (J. Fries)
0000-0003-3433-7245 (J. Fries); 0000-0003-1601-9331 (M. Freund); 0000-0002-0702-510X (A. Harth)

monitor operational data to identify trends, make data-driven decisions, and share their data with internal or external stakeholders while ensuring data security and privacy.

Implementing an IoT system with access control for data exchange requires overcoming several challenges. First, the system must uniquely identify which individual users are authorized to access which data. Second, the system must be able to integrate dynamic data, such as raw sensor data in various formats from different sources, and static data. Finally, the system must be easy to set up and be able to integrate different IoT devices from different vendors using different protocols.

To implement a system that can operate in such a heterogeneous environment, we propose to use the Web of Things (WoT) Architecture [1] and a Solid Personal Online Data Store (Pod) (which is based on LDP [2]) combined with an orchestrator component that identifies and interacts with composite IoT devices. Composites consist of standalone IoT devices. IoT devices are referred to as Things. For easier system integration and interoperability, all Things and their interactions are described with WoT TDs [3]. An alternative approach that implements interoperability and eases system integration is the Asset Administration Shell [4].

A Solid Pod is a data store that resembles a file system, exposes a REST API with controlled access to the data. Solid implements relevant LDP concepts, namely LDP containers, which correspond to file system directories, and (non-container) LDP resources, which correspond to files.

The contribution of this work is an architecture that is (i) extensible in terms of server components, with the orchestrator as a coupling client in between, (ii) eases implementation by using existing server software, and which (iii) builds on established Semantic Web technologies such as WoT and Solid to facilitate system and data integration.

The running example is an industrial machine comprising motors and sensors. A motor moves a conveyor belt, while a sensor is placed next to a conveyor belt to detect workpieces on the belts. Examples for workpiece detection sensors are color sensors or ultrasonic sensors. Each sensor and motor is associated with exactly one conveyor belt. The conveyor belts can be placed in series to move workpieces between locations. A series of conveyor belts is an instance of a composite Thing.

2. Related Work

Ramachandran *et al.* [5] use Solid Pods as decentralized storage for IoT data, without involving the manufacturers of IoT devices. The Pod is used to store data with access control, while a blockchain is used to verify the authenticity of data. IoT devices interact with the Pod directly or through an external service. While the authors use the Pod for a similar purpose, our architecture facilitates access control based on the compositional aspects of IoT devices, and explicitly decouples IoT devices from the Pod to enable extensibility.

Moons *et al.* [6] introduce an approach for storing wireless sensor data in a Solid Pod. The architecture of the authors' approach is based on gateways as termination points for wireless sensor networks (WSNs) and a management server that forwards sensor data. The authors find that the lowest sensor message overhead results in the least battery drain, and thus use another component to map sensor data to RDF. Similar to our approach, each sensor has an LDP resource

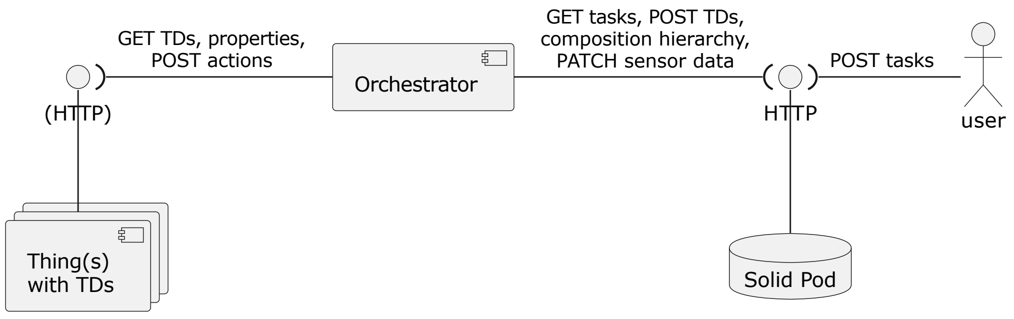

Figure 1: UML component diagram of the proposed architecture.

in the Pod that is updated with new data. The gateways are akin to the orchestrator proposed in our work. Since our architecture covers composite Things with actuators, the orchestrator also offers interactivity and creates a directory structure based on the device composition to enable controlled access to device data.

Bader and Maleshkova [7] introduce SOLIOT as an extension of Solid Pods with CoAP and MQTT. In SOLIOT an LDP container is considered to be the same as a WoT Thing. The authors highlight the advantages of CoAP compared to HTTP. Our paper presents a general architecture that uses off-the-shelf software, where IoT data is accessed via HTTP and WoT Things are not the same as LDP containers. We do not directly couple the Pod to composite IoT devices and vice versa, which enables using unmodified Solid server software. The Things and relations between them are reflected in the Pod, even without coupling.

3. Approach

Fig. 1 shows the proposed architecture, consisting of a composite Thing, an orchestrator and a Solid Pod. The central component is the orchestrator, which controls the composite Thing through the various standalone IoT devices that are part of the composite, but also has read and write access to a Solid Pod.

The following sections describe each component in detail, including what data the component takes as input and produces as output, which interfaces the component must have, and how a component interacts with other parts of the system.

3.1. Standalone Things and Subsystems of the Composite

The lowest-level components of the architecture are the standalone Things, i.e., the sensors and actuators. We assume that these Things are constrained in terms of energy, computation, and storage. In the example of a series of conveyor belts, the standalone Things are motors and sensors. The individual conveyor belts represent the subsystems of the composite series of belts, and each belt comprises a motor and a sensor.

Actuators take commands as input, e.g., run the motor. Sensors and actuators produce output: sensors measure the environment, e.g., a distance value, while actuator output is the current state of the device, like whether a motor is stopped. Such outputs are dynamic Thing data.

In our architecture, we assume that, since standalone Things are the most constrained components, they only interact with one central component, the orchestrator. The orchestrator interacts with the Things to implement specific functionality. Third parties cannot interact with Things directly. This limitation means that dynamic data is only available in real-time on the orchestrator. Sensors and actuators can optionally have associated data that does not change, such as datasheets, manuals, or repair guides, which we refer to as static data. Both static and dynamic Thing data can be linked together and made available to stakeholders as part of our architecture.

The orchestrator is a client that uses the TDs of standalone Things and subsystems to interact with them. In the WoT, interactivity is based on interaction affordances, which are invitations to read, write, or observe `properties`, invoke `actions`, or subscribe to `events` [8]. The TD is vendor-agnostic and protocol-agnostic because interaction affordances can be mapped to multiple protocol bindings. As a result, the interactions between Things and the orchestrator are not limited to specific vendors and are not limited to HTTP. The system can, for example, fulfill Quality of Service requirements through MQTT or OPC UA with TSN. The relationship of a standalone Thing to other Things and subsystems is encoded in TDs using Web Linking [9].

The orchestrator is initially aware of all TDs through some discovery mechanism. Details on the creation and discovery of TDs are out of scope for this short paper.

In our conveyor belt series example, the sensors and motors are connected to an intermediary entity that translates low-level interactions to HTTP. As a result, sensors and motors take the role of an HTTP server, while the orchestrator is an HTTP client. The intermediary enables reading a sensor's current value through a `property`, running the motor through `actions` and reading the current state (as a `property`) of the motor. The sensor and motor TDs reference the corresponding conveyor belt using the `controlledBy` link relation. The link relation leads to a TD that serves as an identifier for the conveyor belts.

3.2. Orchestrator

The orchestrator is a constrained device with limited computing capabilities and storage capacity, but is not energy constrained, i.e., not battery powered. An example of such a device is an embedded or industrial computer. The orchestrator is the central component of the architecture and acts as a client to the standalone Things and to the Solid Pod.

The orchestrator directly interacts with the Pod and Things and is thus referentially (i.e., it needs to be aware of all IoT devices and the Pod) and temporally (i.e., it initiates all communication and waits for responses) coupled to all entities in the architecture. This coupling enables extensibility, since the Pod can change independently of IoT devices and vice versa. Due to the usage of WoT, the system implementer can focus on implementing the domain logic, and network protocol-specific implementation details are abstracted away.

In general, the orchestrator fulfills two roles. The first role is to act as a relay for data generated by standalone Things: the orchestrator reads raw data from standalone Things and writes this data to the Solid Pod via HTTP. Consequently, the data generated by Things is materialized in the Pod. The orchestrator and the Pod are separate hosts in different networks. The orchestrator pushes data to the Pod periodically due to computational and/or connectivity constraints. The static data of the Things, like a PDF-formatted datasheet, is pushed to the

Pod during the initialization process and is therefore available as soon as the orchestrator is initialized.

The dynamic and static Thing data is subject to fine-grained access control. The owner of the machine can give stakeholders access to the data of standalone Things or whole subsystems. To achieve controlled access, the orchestrator creates an LDP container tree on the Pod, where nesting corresponds to the compositional hierarchy, i.e., the relationships between standalone Things and subsystems found in TDs. This LDP container tree is created during the orchestrator's initialization process, i.e., the orchestrator is already aware of all TDs. Each subsystem of the composite has an LDP container with the subsystem's TD, and the standalone Things are the leaves of the LDP container tree. In such a leaf LDP container, the orchestrator pushes the relevant dynamic and static Thing data and corresponding TDs.

To facilitate the processing and integration of the collected dynamic data in the LDP container, the orchestrator maps raw data to RDF before pushing the data to the Pod. Following existing recommendations [8], SOSA/SSN [10] and SAREF [11] are suitable. This mapping enables linking the sensor data to the TDs and to the compositional hierarchy in the Knowledge Graph (KG) that is formed by all the Linked Data in the Pod. Furthermore, static data can be described using RDF for integration into the KG.

In the conveyor belt example, the LDP container tree is instantiated by creating an LDP container for each conveyor belt (`./ConveyorBelt[N]/`). Inside each of those LDP containers, the orchestrator creates two LDP containers, one for the belt's sensor and one for the belt's motor (`./ConveyorBelt0/motor/`). When the LDP container tree is created, the orchestrator pushes all static data into the respective LDP containers. Note that the orchestrator should remove all interaction forms from TDs written to the Pod, as only the orchestrator should be able to interact with standalone Things. The orchestrator pushes dynamic Thing data to the Pod while executing tasks or after a task is finished with HTTP PATCH.

The second role of the orchestrator is to read and execute tasks that were written by stakeholders to a task queue LDP container. The idea of a task queue is similar to an inbox as defined by Linked Data Notifications (LDN) [12]. This approach requires tasks to have a structured format that can be parsed by the orchestrator. The task queue is suitable if tasks are long-running (or task push frequency is low), which makes real-time interaction with the composite Thing unnecessary, as the orchestrator simply executes one task after another without new input.

The orchestrator periodically fetches the latest task and orchestrates the composite based on the task and the orchestration logic. The orchestrator fulfills tasks by interacting with the standalone Things that comprise the composite Thing based on interaction affordances described in TDs. The interaction sequence is implemented as orchestration or control logic. The orchestrator for the conveyor belt example is implemented using the WoT Scripting API [13]. Tasks correspond to stopping points on or between conveyor belts to, e.g., modify a workpiece.

3.3. Solid Pod

The Solid Pod is the only unconstrained component in the architecture. The Pod acts as the data store for all dynamic and static data of the Things and the corresponding TDs. The Pod also hosts the task queue to link the task history to the Thing data. Stakeholders cannot write to the

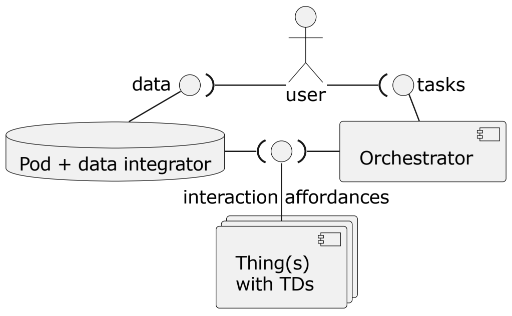

Figure 2: UML component diagram of a virtual integration approach.

task queue or read any dynamic or static data, unless the machine owner adds the stakeholders' WebID to an access control list (ACL). Each LDP container and resource has an ACL, or inherits the ACL from parent LDP containers. ACLs can change over time, enabling the machine's owner to revoke access.

Since the orchestrator is using existing NodeJS libraries[1] to interact with the Solid Pod, the Solid server software needs to be compatible with these libraries. Furthermore, Solid does not provide a built-in mechanism to give users access to only the files that they created, by default stakeholders cannot read their already written tasks. To solve this issue, a task's creator may append their WebID to the task. The orchestrator then modifies the ACL of the corresponding LDP resource. To do so, the orchestrator needs to be able to change ACLs and thus needs full access to the Pod. On an Enterprise Solid Server, client credentials[2] can be used to generate credentials for applications like the orchestrator.

3.4. Enabling Near Real-Time Data

The orchestrator pushes Thing data to the Pod periodically due to resource constraints. Each push incurs network latency (i.e., the delay between orchestrator and Pod) because the Pod is assumed to be unconstrained and hosted on a different network. The sum of network latency and time between pushes provides an upper bound for the staleness of the data.

Removing the influence of the interval between pushes on staleness requires multiple aspects to change. Because the alternative to pushing Thing data is polling, the Pod (acting as a client when polling) and the orchestrator (acting as a server when polling) need to become servients (server and client at the same time). Polling increases implementation complexity, as the orchestrator needs to act as a server and the Pod needs to be modified to act as a client.

The orchestrator periodically polls the task queue on the Pod for new tasks in case the orchestrator is idle. In case the orchestrator is busy, the polling happens immediately after finishing a task. Once a task is being executed, the orchestrator does not poll for new tasks. In the idle case, time to task execution is delayed by the network latency and the polling interval. In the busy case, the time for the remaining tasks and the task fetch time (network latency) delay a new task's execution. An alternative is to push tasks directly to the orchestrator and persisting

[1] https://github.com/inrupt/solid-client-js
[2] https://docs.inrupt.com/ess/latest/services/service-application-registration/

them there. Saving tasks requires the orchestrator to have enough storage for the full task history. This approach also requires orchestrator and Pod to be servients, as the communication pattern is polling instead of pushing.

If the task queue is moved from the Pod to the orchestrator, then the only aspect that still couples Pod and orchestrator together is the Thing data that is exchanged between Pod and orchestrator. Easing constraints further, if the Things can handle a high data query frequency and concurrent queries from both the orchestrator and the Pod, the Pod may directly query the Things and act as a query forwarder that can save data but also fetch the most recent data if requested. Fig. 2 shows this resource-intensive approach, where both orchestrator and Pod must be servients. This approach minimizes delays due to fewer intermediaries between IoT devices and end users (i.e., stakeholders).

3.5. Fine Grained Access Control Use Cases

In the example of an industrial machine, the owner of the machine can naturally access all data. Customers wanting to use the machine, on the other hand, only have high-level access to the task queue. All stakeholders can expect the orchestrator to give read and write access exclusively to the submitter of a task. Another use case is maintenance work being done on the machine by an external party. The owner can give selective access to relevant data to this third party. For example, if a conveyor belt that is part of the machine is broken, the repair process can be made easier by providing access to the motor and sensor data of that conveyor belt. The machine owner simply gives read access to the relevant LDP container. Access can also be revoked at a later time, enabling, for example, temporary authorization based on the duration of contracts.

4. Conclusion and Future Work

This short paper presented an architecture and corresponding implementation that uses Semantic Web technologies to enable access to data of composite machines with fine-grained access control. WoT TDs ease system integration and RDF is used for data integration. An orchestrator component enables access-control on a data store by mirroring the hierarchical relationships between standalone devices and subsystems that form the composite to the data store. The owner of the machine can give specific stakeholders access to relevant parts of the data based on the hierarchy. The architecture provides a mechanism to enable stakeholders to interact with the composite machine. The composite Thing and the data store can both evolve independently, as they are decoupled. However, the orchestrator, which implements control functionality and is responsible for moving data, is coupled to the composite Thing and the Pod. Future work includes applying the architecture to more use cases, evolving the architecture, and an empirical evaluation of the performance characteristics beyond theoretical latency bounds.

Acknowledgements

This work was funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) through the Antrieb 4.0 project (Grant No. 13IK015B).

References

[1] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, K. Kajimoto, Web of Things (WoT) Architecture, Recommendation, W3C, 2020. https://www.w3.org/TR/2020/REC-wot-architecture-20200409/.

[2] S. Speicher, J. Arwe, A. Malhotra, Linked Data Platform 1.0, Recommendation, W3C, 2015. https://www.w3.org/TR/2015/REC-ldp-20150226/.

[3] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, M. Kovatsch, Web of Things (WoT) Thing Description, Recommendation, W3C, 2020. https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/.

[4] S. Bader, E. Barnstedt, H. Bedenbender, B. Berres, M. Billmann, M. Ristin, Details of the Asset Administration Shell - Part 1 (2022). https://www.plattform-i40.de/IP/Redaktion/DE/Downloads/Publikation/Details_of_the_Asset_Administration_Shell_Part1_V3.html.

[5] M. Ramachandran, N. Chowdhury, A. Third, Z. Jan, C. Valentine, J. Domingue, A Framework for Handling Internet of Things Data with Confidentiality and Blockchain Support, in: Proceedings of the Workshop on IOT Infrastructures for Safety in Pervasive Environments (IOT4SAFE 2020), Herakleion, Greece, June 2, 2020, volume 2686 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2020. URL: http://ceur-ws.org/Vol-2686/paper5.pdf.

[6] B. Moons, F. Sanders, T. Paelman, J. Hoebeke, Decentralized Linked Open Data in Constrained Wireless Sensor Networks, in: 7th International Conference on Internet of Things: Systems, Management and Security, IOTSMS 2020, Virtual Event, France, December 14-16, 2020, IEEE, 2020, pp. 1–6. URL: https://doi.org/10.1109/IOTSMS52051.2020.9340221.

[7] S. R. Bader, M. Maleshkova, SOLIOT - Decentralized Data Control and Interactions for IoT, Future Internet 12 (2020) 105. URL: https://doi.org/10.3390/fi12060105.

[8] V. Charpenay, S. Käbisch, On modeling the physical world as a collection of things: The W3C thing description ontology, in: The Semantic Web - 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31-June 4, 2020, Proceedings, volume 12123 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 599–615. URL: https://doi.org/10.1007/978-3-030-49461-2_35.

[9] M. Nottingham, Web Linking, RFC 8288 (2017) 1–24. URL: https://doi.org/10.17487/RFC8288.

[10] A. Haller, K. Janowicz, S. J. Cox, M. Lefrançois, K. Taylor, D. Le Phuoc, J. Lieberman, R. García-Castro, R. Atkinson, C. Stadler, The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation, Semantic Web 10 (2019) 9–32.

[11] L. Daniele, F. T. H. den Hartog, J. Roes, Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology, in: Formal Ontologies Meet Industry - 7th International Workshop, FOMI 2015, Berlin, Germany, August 5, 2015, Proceedings, volume 225 of *Lecture Notes in Business Information Processing*, Springer, 2015, pp. 100–112. URL: https://doi.org/10.1007/978-3-319-21545-7_9.

[12] S. Capadisli, A. Guy, Linked Data Notifications, Recommendation, W3C, 2017. https://www.w3.org/TR/2017/REC-ldn-20170502/.

[13] Z. Kis, D. Peintner, C. Aguzzi, J. Hund, K. Nimura, Web of Things (WoT) Scripting API, W3C Note, W3C, 2020. https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/.