Adaptable Interfaces, Interactions, and Processing for Linked Data Platform Components

Felix Leif Keppmann
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
felix.leif.keppmann@kit.edu

Maria Maleshkova
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
maria.maleshkova@kit.edu

Andreas Harth
Karlsruhe Institute of Technology
Kaiserstraße 12
76131 Karlsruhe, Germany
andreas.harth@kit.edu

ABSTRACT

Currently, we are witnessing the rise of new technology-driven trends such as the Internet of Things, Web of Things, and Factories of the Future that are accompanied by an increasingly heterogeneous landscape of highly modularized devices and pervasion of network-accessible "things" within all areas of life. At the same time, we can observe increasing complexity of the task of integrating subsets of heterogeneous components into applications that fulfil certain needs by providing value-added functionality beyond the pure sum of their components. Enabling integration in these multi-stakeholder scenarios requires new architectural approaches for adapting components, while building on existing technologies and thus ensuring broader acceptance. To this end, we present our approach on adaptation, that introduces adaptable interfaces, interactions, and processing for Linked Data Platform components. In addition, we provide an implementation of our approach that enables the adaptation of components via a thin meta-layer defined on top of the components' domain data and functionality. Finally, we evaluate our implementation by using a distributed benchmark environment and adapting interfaces, interactions, and processing of the involved components at runtime.

CCS CONCEPTS

• **Theory of computation → Semantics and reasoning**; **Program semantics**; • **Computing methodologies → Distributed artificial intelligence**; • **Social and professional topics** → Software selection and adaptation;

KEYWORDS

Read-write Linked Data, Linked Data Platform, Distributed Applications, Distributed Control, Integration, Adaptation

ACM Reference format:
Felix Leif Keppmann, Maria Maleshkova, and Andreas Harth. 2017. Adaptable Interfaces, Interactions, and Processing for Linked Data Platform Components. In *Proceedings of Semantics2017, Amsterdam, Netherlands, September 11–14, 2017,* 8 pages.
https://doi.org/10.1145/3132218.3132243

1 INTRODUCTION

Driven by current technology developments, we are witnessing the increasing popularity of new trends such as the Internet of Things (IoT) [2], Web of Things (WoT) [6], Semantic Web of Things (SWoT) [9], Factory of the Future, or "Industrie 4.0", which are accompanied by an increasingly heterogeneous landscape of small, embedded, and highly modularized devices and applications, multitudes of manufactures and developers, and pervasion of network-accessible "things" within all areas of life. These developments are tightly associated with the growing complexity of handling the integration of heterogeneous components as part of distributed applications, which fulfil certain needs by providing value-added functionality based on the collaboration of involved components. We notice these integration difficulties not only at data and protocol level, but also at application level.

Coping with diverse multi-stakeholder data integration scenarios is a known challenge, which is already addressed by several approaches tackling the related issues. The Linking Open Data Cloud (LODC) [1] is one example for multi-stakeholder data integration. Following the Linked Data (LD) principles [4], Linked Data enables integration at semantic level, data model level, as well as protocol level. However, it is primary targeting the integration of read-only datasets. In this context, the Linked Data Platform (LDP) recommendation [16] of the World Wide Web Consortium (W3C) combines the Linked Data principles with the Representational State Transfer (REST) paradigm [7] and specifies read-write Linked Data resources and containers.

While these approaches focus on the integration from a data-centric viewpoint, some important aspects of integration in the context of distributed applications still remain unaddressed. In particular, we focus on the challenges that result from creating composite applications from a set of components, which have different functionality and which transfer data between each other through interactions over a network. With respect to integration scenarios, in which multiple stakeholders are independently involved in the development and deployment of distinct components, we face several integration challenges. We discuss these challenges in detail in three problem areas below.

P1: Data Models and Protocols. One of the main integration burdens have been missing or ambiguous data semantics as well as incompatible data models, data formats, and protocols. With increasing modularization, these burdens become more visible, for example, as it is the case during the integration of smart home sensors, classical web services, household appliances, and mobile devices into a single home automation application. This situation is aggravated when components originate from different domains and manufactures. In these cases, we are challenged with the creation of a common integration architecture that enables the collaboration of all components, including mappings for compatibility between data models and protocols. However, the enforcement of the one integration architecture on every device may not be the best solution and even reduce the overall functionality. For instance, certain protocols and data formats might suit local conditions considerably better than the integration architecture, e.g., wireless transmission protocols with low energy and computing power consumption. In these cases, dedicated components may be required as proxies between domain-specific architectures and the integration architecture.

P2: Interaction and Processing Patterns. Further challenges for integration are different interaction and processing patterns. On the one hand, data flows between components can be established through complimentary interaction patterns. For instance, components provide interfaces and other components receive data as the payload of answers to their requests at these interfaces. In contrast, other components provide interfaces and data is transferred as payload of requests of components to these interfaces. On the other hand, requests that transfer data to or from other components and the provisioning of data at interfaces can be initiated by different processing patterns. For example, a sensor measures the temperature at a fixed frequency, processes the results, and provides the data at an interface or transfers the data to other components. In contrast, a sensor measures the temperature only if other components request the data at the interface. Integrating several components with diverting requirements on interaction and processing patterns into one distributed application is challenging.

P3: Knowledge about Integration Scenarios. With respect to integration scenarios, in which multiple components are included in distributed applications, we face a challenging lack of knowledge. In particular, the more distinct and smaller components become, the more stakeholders are involved in their development, and the more broadly usable and generic their provided functionality is. As a result, during the design of the components there is little or no knowledge about possible future integration scenarios, i.e., it is unknown 1) which subset of data with which semantic annotations 2) must be provided by interfaces or 3) must be transferred to or from other components through interactions.

With respect to these challenges, we make the following contributions:

C1: We present our **approach on adaptation**, a refined and improved version of our Smart Component (SC) [10] approach, that is based on well-established integration paradigms, provides requirements to cope with the challenges, and includes our architecture for adaptable Linked Data Platform components, that satisfies these requirements.

C2: We provide the Smart Component Adaptation Layer (SCAL) as **implementation of our approach** that provides the adaptation capabilities of our architecture through a thin layer on top of or integrated with domain-specific functionality of components.

C3: We conduct an **evaluation of our implementation** by enabling centralized and decentralized query evaluation settings for the Distributed LUBM (DLUBM), a truly distributed Linked Data benchmark environment, through adaptations with SCAL.

2 ADAPTABLE LINKED DATA INTERFACES, INTERACTIONS, AND PROCESSING

2.1 Definitions

With respect to our viewpoint on distributed applications, we define different core concepts that provide a certain level of abstraction for generalization but at the same time keep important characteristics related to the challenges.

D1: Functionality Functionalities are domain-specific functions and data that we keep as black boxes. Functionalities provide internal data, or require external data to work, or provide as well as require data. Thereby, we abstract away from specialized domain-specific issues and focus on general integration challenges.

D2: Component. Components are the building blocks of integration that provide functionality at the network. Components establish data flows between each other through pull or push interaction. Therefore, components 1) expose interfaces for requests, i.e., take on the server role, or 2) execute requests at interfaces, i.e., take on the client role, or 3) take on both roles at the same time. Components execute the requests that transfer payloads to or from other components and the steps to provide data at interfaces for other components with active or passive processing. Therefore, components 1) actively initiate the processing by themselves, or 2) passively trigger the processing on events caused by other components, or 3) combine passive and active processing.

D3: Application. Applications are compositions of components that combine the distinct functionalities of participating components through data flows to provide a value-added functionality. At least one component per distributed application must be actively processing and executing interactions, while other components may passively react, in order to run a distributed application.

D4: Lifecycle. Components and applications follow separate but interwoven lifecycles. First, components are independently designed, deployed, running, and participating in applications. Second, applications are designed, deployed, and running. In the latter case, design denotes the selection and composition of components, deployment denotes the composition-specific adaptation of components, i.e., adaptation of interfaces, interactions, and processing, and running denotes the established data flows between participating components, i.e., provisioning of the value-added functionalities.

2.2 Integration Paradigms

As preliminaries, we build on established architectural paradigms and principles for solving some of the challenges that we identified. In particular, we incorporate the REST and Linked Data paradigms as the foundation of our approach.

Linked Data. The Linked Data paradigm is in a nutshell described by the four well-known Linked Data principles [4]:

(1) "Use URIs as names for things."
(2) "Use HTTP URIs so that people can look up those names."
(3) "When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL)."
(4) "Include links to other URIs, so that they can discover more things."

Representational State Transfer. The REST [7] paradigm introduces a set of architectural constraints that restrict the degrees of freedom for interfaces as well as interaction with these interfaces and, thereby, ease the integration. The Richardson maturity model [23] describes different levels of support for REST:

(0) No support, request are tunnelled through the protocol.
(1) Resources are distinguished and identified by Uniform Resource Identifiers (URIs).
(2) Hypertext Transfer Protocol (HTTP) verbs enable resource access and manipulation.
(3) Embedded Hypermedia controls relate interface parts.

While REST is architecture-agnostic in general, we use the variant built on web technologies such as URIs and HTTP as implied by the maturity model.

2.3 Requirements

In the following, we derive a set of requirements following from the challenges.

R1: Compliance with Integration Paradigms. Our first requirement, with respect to the challenges related to *Data Models and Protocols (P1)* is the compliance with established integration paradigms. Thus, we require compliance of interfaces and interactions with the W3C Linked Data Platform (LDP) [16] recommendation, that specifies for the first time the integrated use of both paradigms Linked Data and REST. The specification handles provisioning of resources adhering to the Resource Description Framework (RDF), of non-RDF resources, and of container resources, a sub-concept of RDF resources for resource collections. The LDP specifies also how clients must interact with these resources. In particular, the REST paradigm enforces HTTP as true application protocol, i.e., by enabling resource-oriented interfaces for applications that support the semantics of HTTP methods and status codes. As a consequence of the Linked Data principles, we require the use of RDF as the primary data model for semantic annotation of data that is provided by or sent to interfaces. However, we do not prohibit the use of specialized data models and formats, described and linked from RDF resources, which is supported by the LDP recommendation in the form of non-RDF resources.

R2: Adaptation of Interfaces, Interactions, and Processing. Our second requirement, with respect to the specific challenges related to the *Interaction and Processing Patterns (P2)*, is the adaptation of interfaces, interactions, and processing.

First, we must be able to adapt the interfaces of components that are part of a composition. This includes: the number, structure, and identifiers of resources, the relevant data to be provided per resource, and the semantic annotations of the data. In addition, this also includes the handling of incoming data with respect to the functionality of the components.

Second, we must be able to adapt components with respect to the interactions with other interfaces in the composition. In the case of push communication, this includes: the selection of relevant data as payload of interactions, the semantic annotations of the data, the identifiers of remote interfaces, and the methods to be used for interaction with these interfaces. In the case of pull communication, this includes: the identifiers of remote interfaces, the methods to be used for interactions, and the handling of incoming data with respect to the functionality of components.

Third, we must be able to influence how components are processing the data, in accordance with their interfaces and interactions. This includes: the definition of triggers for active processing based on time or external events, and the definition of triggers for passive processing based on events related to other components. Please note that we do not require – but also do not prohibit – the processing within the functionality to be adjustable in the same way. Thus, the processing of data for interface updates or execution of interactions may be coupled or decoupled from the processing of the functionality.

R3: Separation of Design, Adaptation, and Runtime. Our third requirement, with respect to the challenges related to the *Knowledge about Integration Scenarios (P3)*, is the separation of design, adaptation, and runtime.

First, we require that domain functionality of components can be developed and implemented during design time. In particular, integration requirements should not force the design to be tailored and restricted to individual scenarios but be focused on the domain functionality with broad applicability. Since every single requirement for integration with other components is, in general, not known at design time, the implementation should provide means for enabling adaptations to these after development.

Second, we require that adaptations can be declared during deployment or, as advanced requirement, be declared during the runtime of components. In the latter case, means for adaptation should adhere to the same integration paradigms chosen as preliminaries. The adaptations must be separated from the design in order to support the adaptation of components during the integration with other components in specific integration scenarios, which might be unknown at design time. Thereby, no modifications of the implementation of domain functionality should be required to support multi-stakeholder situations, in which the implementation of a component is not within the reach of the integrating stakeholder.

Third, we require that processing, passive or active, is separated from the actual adaptations. Adaptations may contain information about processing details but should not initiate processing via their deployment. Thereby, we enable a priori planning, creation, and deployment of adaptations and explicit transition to participation of components in distributed applications, i.e., the runtime of the distributed applications. In addition, separate sets of adaptations for participation in different distributed applications may be supported, for which processing can be individually started or stopped. Hereby, we introduce separation of concerns with respect to their integration requirements.

2.4 Architecture

In the following, we introduce our architecture, shown in Figure 1, that enables adaptable interfaces, interactions, and processing for Linked Data Platform components, and satisfies our requirements.

A1: Functionality, Interpreter, and Interface. In our architecture diagram in Figure 1, we distinguish between three major parts: functionality, interpreter, and interface. With the functionality part, we denote all local domain-specific data and functionality of the components. Components may be small, e.g., temperature sensors, or large, e.g., data stores. In addition, components may integrate other domain-specific interfaces and interactions, but,

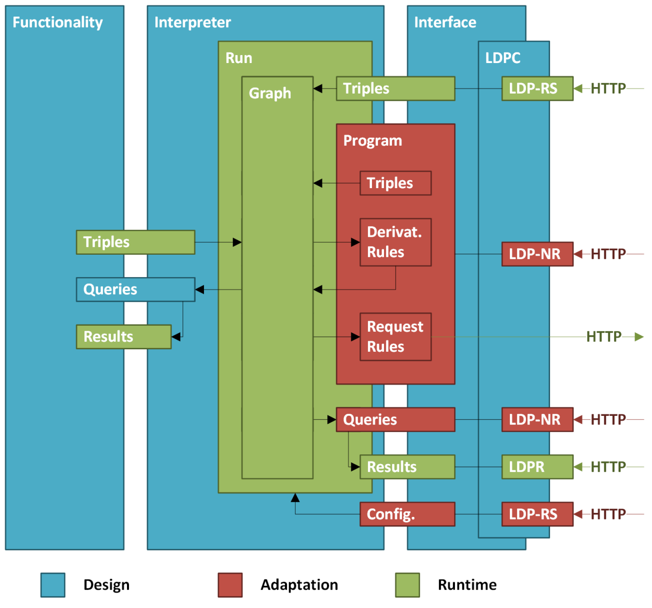

Figure 1: Adaptable LDP Component Architecture

with respect to our chosen integration architecture, only the shown details are relevant. The only requirement is the lifting and lowering of data, i.e., provisioning of relevant information as RDF to the interpreter in case of lifting and provisioning of queries, declared in the SPARQL Protocol and RDF Query Language (SPARQL), to the interpreter as well as processing of their results in case of lowering.

With the interpreter part, we denote an evaluation engine that supports the evaluation of Notation3 (N3) [3] rule programs and SPARQL queries. During every evaluation run, the engine maintains an internal RDF graph that is enriched with triples provided by the domain functionality, by the interface, or by the rule programs. Rules contained in N3 rule programs are iteratively evaluated against the internal RDF graph until a fix point is reached, i.e., until an enrichment of the graph leads to no additional execution of rules. SPARQL queries are evaluated against the state of the RDF graph at the fix point and their results are propagated to the local domain-specific functionality or to the interface.

With the interface part, we denote an LDP-compliant Application Programming Interface (API) that is exposed by components to the network. At the root, i.e., the entry URI of the interface, a Linked Data Platform Container (LDPC), e.g., a Linked Data Platform Basic Container (LDP-BC), forms the base container for the creation, manipulation, and deletion of resources via HTTP interactions over the network. Thereby, we enable the creation and manipulation of triples, programs, queries, and configurations, as well as access to query results. These are each represented by a Linked Data Platform Resource (LDPR), depending on their data model as Linked Data Platform RDF Source (LDP-RS) or Linked Data Platform Non-RDF Source (LDP-NR), and are members of the base container or optional subordinated containers. With respect to provided resources, our architecture does not prohibit but also does not require to have additional LDPRs directly offered by domain functionalities, i.e., additional static interfaces.

As defined in our architecture, components adhering to it provide interfaces to the network that are, by default, compliant with the LDP recommendation. Thereby, we satisfy the first requirement: *Compliance with Integration Paradigms (R1)*. Since the integration paradigms Linked Data and REST, incorporated by the LDP recommendation, have been chosen in the preliminaries as common integration architecture, we, thereby, provide a solution for challenges of the first problem area: *Enclosing Data Models and Protocols (P1)*.

A2: Triples, Programs, Queries, and Runs. In our architecture diagram in Figure 1, we utilize triples and queries for integrating functionality and interpreter, as well as triples, programs, queries, and configurations for adaptation of the interpreter and integration with other components.

To prevent confusion with the internal RDF graph of each interpreter run, we denote further RDF graphs as triples. Relevant information about the current state of the component is provided as triples by the domain functionality. Furthermore, additional information adhering to the RDF data model and can be created as LDP-RS at the LDP interface, i.e., can be created by using a content type of a RDF serialization format.

With SPARQL queries, we provide the means for selecting subsets of information stored in the internal RDF graph of the interpreter at the end of each evaluation, i.e., at the fix points. We enable the selection of information by the domain functionality that is relevant for the functioning of the component. In addition, we enable the creation of queries as LDP-NR at the LDP interface. In this case, query results are exposed as LDPR, more precisely as LDP-NR for table-based content types or as LDP-RS for RDF results.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix http: <http://www.w3.org/2011/http#> .
@prefix http-m: <http://www.w3.org/2011/http-methods#> .
@prefix ex: <http://example.local/vocab#> .

{ ex:foo ex:bar ?object . } => {
  ex:foooo ex:barrr ?object .
} .

{ ex:foo ex:bar ?object . } => {
  [] http:mthd
       http-m:PUT ;
       http:requestURI <http://example.local/resource> ;
       http:body { <> ex:foobar ?object . } .
} .
```

Figure 2: Derivation and Request Rule Example

By supporting the interpretation of N3 rule programs, created as LDP-NR at the LDP interface, we add important adaptation capabilities. First, as a superset of RDF, N3 programs may contain static triples, e.g., configuration settings, which are added during each interpreter run to the internal RDF graph. Second, with derivation rules we enable enrichment of the internal RDF graph. Thereby, we can transform annotations, conduct reasoning, or add complete entailment rule sets, e.g., for the Resource Description Framework Schema (RDFS) or for the OWL-LD subset of the OWL Web Ontology Language (OWL). Third, with request rules, we treat rule heads containing information about HTTP interactions, annotated with the W3C HTTP Vocabulary [11], differently from other rules. Instead of enriching the internal RDF graph, the described HTTP interactions are executed at the network and the payloads of the answers are added to the internal RDF graph. Thereby, we enable the declaration of HTTP interactions as part of N3 rule programs. In Figure 2, we show simple examples for a derivation as well as an interaction rule. Optionally, built-in functions to ease certain tasks may be provided, e.g., mathematical calculations without using external services.

Finally, run configurations, “Config.” in Figure 1, relate the aforementioned concepts in one LDP-RS. Every run configuration declares interpreter settings, programs and queries to be evaluated, resources for query results, and triggers. With the concept of triggers, we provide the means for specifying when the interpreter should evaluate programs and queries. We distinguish between time-base (e.g., frequency, or delay) and event-based (e.g., on resource requested, on resource expired, or on resource changed) triggers. Aligned with our prototypical implementation, presented in Section 2.5, we are developing an adaptation ontology for the specification of run configurations. Due to space constrains, the ontology is not described in detail in this work. In short, all aforementioned concepts can be annotated to declare run instances in RDF. Run configuration are adhering to the RDF data model and can be created as LDP-RS at the LDP interface of the component with aforementioned content types.

With the here described adaptation capabilities, we satisfy the second requirement: *Adaptation of Interfaces, Interactions, and Processing (R2)*. As we enable adaptation of interfaces with queries, adaptation of interaction with request rules, and adaptation of processing patterns with triggers, we, thereby, provide a solution for the challenges of the second problem area: *Interaction and Processing Patterns (P2)*.

A3: Design, Adaptation, and Runtime. In our architecture diagram in Figure 1, we used colour-coding to assign all concepts to corresponding distinct steps of the evolution of adaptable components: blue concepts are assigned to design, red concepts are assigned to adaptation, and green concepts are assigned to runtime.

First, design concepts, i.e., blue colour, are designed and implemented by the original manufacturers and developers of the components. These are, in particular, the domain functionality of the component, the integration of interpreter with the aforementioned domain functionality, and access to the adaptation capabilities of the interpreter through provisioning of the LDP interface.

Second, adaptation concepts, i.e., red colour, are deployed by integrating stakeholders with respect to distributed applications based on the requirements of their specific integration scenarios. On the one hand, for adaptation through LDP interfaces, components are deployed and must be running to provide their interfaces. In this state, the components themselves are at runtime, but not yet part of any distributed application. Once the interpreter runs are started, the components are, passively or actively, part of the runtime of respective distributed applications. For greater usability, adaptations may be stored in a persistent manner, i.e., adaptations survive stops and restarts of components. This will, on the other hand, enable deployment of adaptations as part of the deployment of components, i.e., adaptations are added a priory to storages of components and are already available when components are started. However, our architecture does not specify or prohibit persistent storage of adaptations.

Third, runtime concepts, i.e., green colour, are available during the interpreter runs, i.e., while components are participating in distributed applications via interaction with other components, declared as request rules during adaptation, or by providing interfaces to other components, declared as queries during adaptation. Above, we described the declaration of these adaptations. However, we did not specify the transition from these adaptations to actual interpreter runs, i.e., participation in distributed applications. To keep compliance with the LDP recommendation, we enable transitions from declared run configurations to run instances by executing the HTTP POST method on run configuration resources. In particular, the LDP recommendation states for LDPRs with respect to support of the HTTP POST method[1]:

> "Per [RFC7231], this HTTP method is optional and this specification does not require LDP servers to support it. When a LDP server supports this method, this specification imposes no new requirements for LDPRs."

As stated by the LDP recommendation, HTTP POST is explicitly not restricted for LDPRs, and, thereby, not for LDP-NRs and LDP-RSs, but only for LDPCs and its variants. For LDPCs, the recommendation specifies manipulation of resource collections through HTTP POST. An interpreter run is instantiated based on a run configuration if the HTTP POST method was executed at a LDP-RS, the LDP-RS contains a run configuration, and the contained run configuration is valid. An interpreter run is terminated if the LDP-RS representing the run instance is deleted by executing the HTTP DELETE method. The location of this resource, i.e., its identifier, is part of the run configuration. For increased usability, the run state of interpreters may be part of optional persistence and, thereby, enable recovery of states, e.g., automatic restarts of runs after stops and restarts of components.

With the aforementioned three steps, starting with the design of components, moving on to the adaptation of components, and up to the runtime of distributed applications, we satisfy the third requirement: *Separation of Design, Adaptation, and Runtime (R3)*. Since every step can be individually handled, though still building upon each other, we also provide a solution for challenges of the third problem area: *Knowledge about Integration Scenarios (P3)*.

2.5 Implementation

With the Smart Component Adaptation Layer (SCAL)[2], we provide a prototypical implementation of our approach. SCAL is, on the one hand, a file system-based LDP server and, on the other hand, supports adaptation by integrating a N3 interpreter and a Linked Data query engine.

Linked Data Platform Server. Our server implementation is based on the file system as storage backend and supports creation and modification of LDP-RS, LDP-NR, and LDP-BC as well as nesting of containers. Internally, the resources are managed independently from the LDP interfaces and support programmatic integration with other software libraries. A processing manager handles

[1]https://www.w3.org/TR/ldp/#h-ldpr-http_post
[2]https://github.com/fekepp/scal

registration of event listeners and, thereby, enables, e.g., the processing of HTTP POST request by listeners, or the registration of on-request triggers.

Linked Data-Fu Interpreter. As interpreter, we build on Linked Data-Fu (LD-Fu)[3] [18, 20], an interpreter for N3 rule programs with SPARQL query capabilities. LD-Fu supports our requirements with respect to the evaluation of N3 rule programs and SPARQL queries, provides means for executing request rules, and supports mathematical operations with built-in functions. The interpreter is internally managed by a run manager that handles incoming HTTP POST requests on run declarations, configures and initialises the interpreter, and manages the execution of runs with time-based as well as event-based triggers.

Integration. For integration, we support running SCAL standalone, i.e., integrating solely through HTTP requests, integration with domain functionality through the storage subsystem, or, as the most efficient option, programmatic integration as software library. In the latter case, we support the direct programmatic registration of RDF sources and SPARQL queries by the domain functionality.

3 EVALUATION

We evaluate our approach by utilizing the SCAL implementation to integrate multiple components in a Linked Data benchmarking scenario. In particular, we use the Distributed LUBM (DLUBM), a distributed benchmark environment for Linked Data query engines, deploy query evaluation in a centralized as well as two decentralized ways, and compare the query evaluation performance in these settings. In the following, we first introduce the DLUBM, then describe our experiments, and finally discuss the results.

3.1 Distributed LUBM

The Distributed LUBM (DLUBM)[4] is a distributed benchmark environment for Linked Data query engines. Using an extended version of the Lehigh University Benchmark (LUBM) data generator, DLUBM generates distinct but interlinked RDF graphs that are distributed to multiple hosts and supports automated deployment as well as reproducibility of the benchmark environment. An instance of the DLUBM benchmark environment can be described by a set of parameters that influence the structure, data generation, and distribution of the environment. We show an abstract overview of the DLUBM structure at the top left side in Figure 4. In general, the environment simulates a university scenario and provides three levels of granularity – global, university, and department – where every component provides a distinct graph of the overall dataset, including links to related graphs. One component provides a global graph that contains links to graphs which represent universities and that are hosted by independent components, which again contain links to graphs that represent the departments of the universities and which are also hosted by independent components. Department graphs contain the majority of information, e.g., about professors, courses, or students, and may contain links to graphs of other universities. SPARQL queries, provided by the original LUBM benchmark, can be adjusted to and evaluated against the interlinked DLUBM benchmark environment and require Linked Data query engines capable of following and resolving links as well as evaluating SPARQL queries against returned RDF.

[3]https://linked-data-fu.github.io
[4]https://github.com/fekepp/dlubm

3.2 Experiments

Amount	Type	OS	Description
1	m3.large	Ubuntu 17.04	Experiments
1	m3.large	RancherOS 1.0.2	Master
50	m3.large	RancherOS 1.0.2	Worker

Figure 3: AWS EC2 Instances for Experiments

Our experimental setup consists of 1) a DLUBM instance and 2) an infrastructure that provides computing resources to deploy the environment. A DLUBM instance can be described by a set of parameters such as *DLUBM(seed, granularity, university offset, university amount, university limit, department limit)*. We use the configuration *DLUBM(0, DEPARTMENT, 0, 5, 1, 1)*. In detail, this configuration initializes all data generators with a seed of "0", the "DEPARTMENT" granularity leads to the generation of components on all three levels, the "university offset" lets the data generation start with the first university, the "university amount" limits the generation to five universities, i.e., the scale of the environment, and the "university limit" and "department limit" lead to provisioning of a single university or department graph per component. In sum, this configuration leads to 100 components providing LUBM Linked Data, i.e., components of our distributed evaluation application.

The deployment automation for the DLUBM benchmark environment is achieved through container-based virtualisation, in particular, through the Docker[5] ecosystem. We provide a Docker container declaration[6] as well as a Docker image[7] that merges DLUBM and SCAL by replacing the default web server with our SCAL implementation. Thereby, we enable common access to the interlinked DLUBM datasets, while at the same time provide adaptation capabilities.

Instances of the DLUBM benchmark environment can be deployed with Docker tooling to various computing resources, e.g., local computers, private clouds, or Platform as a Service (PaaS) solutions. For comparability, we deploy our DLUBM instance via Docker Swarm mode on computing resources provided by the Elastic Compute Cloud (EC2) of Amazon Web Services (AWS). In detail, as shown in Figure 3, we use one "m3.large" EC2 instance with Ubuntu as operating system for experiments and a set of one "m3.large" manager and 50 "m3.large" worker EC2 instances with RancherOS as Docker-centric operating system for assembling a Docker swarm. Thereby, every EC2 instance hosts in average two DLUBM SCAL containers.

3.3 Measurements

We used our experimental setup to evaluate three different query evaluation settings: centralised query evaluation at global level, decentralised query evaluation at university level, and decentralised query evaluation at department level. In particular, we omit in our experiments the evaluation of queries by an external Linked Data

[5]http://www.docker.com
[6]https://github.com/fekepp/dlubm-scal
[7]https://hub.docker.com/r/fekepp/dlubm-scal/

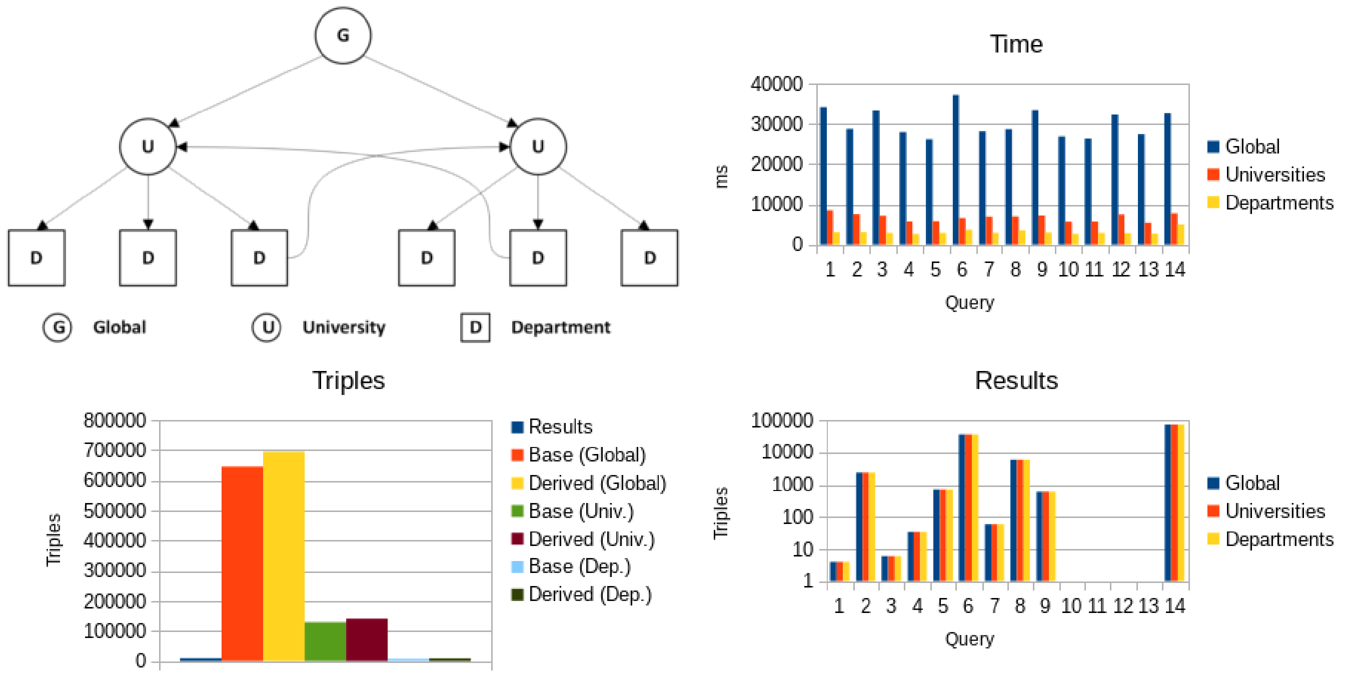

Figure 4: DLUBM Structure and Evaluation Results

query engine but let the components take over this evaluation. For every query, the components must get all relevant graphs that are required for the evaluation, derive additional information with entailment rule sets, in this case the RDFS entailment rule set, evaluate the query, and provide a resource with the result that can be pulled by clients. The query evaluation is triggered on-request to allow measurement of the evaluation with respect to time. We provide a repository[8] for the evaluation, including the DLUBM SCAL, the experiments, as well as our results used in this work. The adaptations are deployed at the LDP interfaces of components after the DLUBM SCAL environment is started. Due to space constrains, we do not list all adaptations of components in detail.

In Figure 4, we provide an overview of the results of our measurements. We measured, with respect to all three evaluation settings, i.e., global, university, and department, the overall evaluation time, the average amount of triples added by request rules to the internal RDF graph of components' interpreters, and the number of results retrieved by a client. Starting with the latter one, we see equal numbers of query results for all scenarios. This is a desired behaviour, since in all scenarios the results must be the same, independently whether we have a global evaluating interpreter or merged results of the interpreters of several components. However, we need to note that the LUBM queries allow separate evaluation. The discussion of completeness is out of the scope of this work, but due to the RDFS entailment rules, that are used by all components, we see results for all queries, except for the more sophisticated queries 10-13.

The overall evaluation time, however, differs significantly between the scenarios. The results confirm the expectation that splitting evaluation to multiple components of equal computing power leads to faster overall evaluation time. While the evaluation at the global component takes in average 30.29 seconds, we see significantly faster evaluation with 6.78 seconds at university level, and 3.19 seconds if queries are evaluated at department level.

The slower evaluation at a single global component is not caused by the requests that transfer all university and department data to the component, but is rather caused by deriving new triples with the RDFS entailment rules. This is shown in the triples diagram, where the number of result triples and the average amount of requested (base) as well as the average amount of derived triples are compared. The more distributed the evaluation is, the fewer derived triples must be computed per component.

Summarizing our experiments, we have shown that our approach and implementation allow the adaptation of components at runtime to the requirements of specific integration scenarios, in this case three different Linked Data benchmarking scenarios. Furthermore, we have shown that our approach is scalable by deploying these integration scenarios as multi-component and heavily distributed applications to computing resources of a PaaS provider. Finally, our performance analysis provides some insights about the implementation characteristics and shows some trade-offs between applications that are centralized and thus easier to manage, and distributed but also more complex to deploy.

4 RELATED WORK

Semantic approaches have already been developed and applied in the context of adaptation for distributed solutions for both data and applications. In this context, related work can be split into three main areas: 1) distributed Read-Write Linked Data (RWLD), 2) creating composite applications based on Web of Things (WoT)

[8]https://github.com/fekepp/dlubm-scal-eval

technologies, and 3) applications based on the Semantic Web of Things (SWoT).

Aligned with our integration preliminaries (Section 2.2) Read-Write Linked Data builds on the idea of combining the architectural paradigms of Linked Data [4] and REST [7]. This combination has been used in several approaches, e.g., Linked Data Fragments (LDF) [21], Linked APIs (LAPIS) [19], Linked Data Services (LIDS) [17], RESTdesc [22], or Linked Open Services (LOS) [12]. As already mentioned, standardization efforts for the integrated use of Linked Data and REST led to the LDP [16] W3C recommendation. Furthermore, Linked Data in combination with REST is used for the foundation of a number of solutions for exposing access to data or creating query interfaces based on SPARQL queries. For instance, grlc [13], evolving on top of tools such as BASIL [5], provides a small server for automatically converting SPARQL queries into Linked Data APIs.

The Web of Things (WoT) [8] builds on top of the IoT in order to provide integration of devices, applications, objects, i.e., "things", not only on the network layer, i.e., the internet, but also on the application layer, i.e., the web. This can be achieved by making things part of the web by providing their capabilities as REST services, based on URIs for identification and HTTP as application protocol for transport and interaction. Integrating these technologies has been, for example, addressed for embedded devices in [6].

In order to foster horizontal integration and interoperability, the Semantic Web of Things (SWoT) [9] focuses the common understanding of multiple capabilities and resources towards a larger ecosystem by introducing Semantic Web technologies to the IoT. Challenges related to SWoT have been, for example, addressed by the SPITFIRE [14] project, or the Micro-Ontology Context-Aware Protocol (MOCAP) [15], both in the area of sensors. We build upon several synergies introduced by a common resource-oriented viewpoint of the Linked Data and REST paradigms. These paradigms also play a key role in WoT and in particular SWoT to cope with heterogeneous data models and interaction mechanisms. In this context, our approach aims to enable the adaptation of components to specific application scenarios at runtime, while still being compatible with other approaches based on Read-Write Linked Data resources.

5 CONCLUSION

New technology trends such as the IoT, WoT, Factories of the Future, and "Industrie 4.0" lead to an increase in the complexity of handling the integration of heterogeneous components as part of distributed applications. We witness these integration difficulties not only at data and protocol level, but also at application level. Therefore, supporting integration in multi-stakeholder scenarios requires new architectural approaches for adapting components, while building on existing technologies and thus ensuring broader acceptance. In this context, we present our approach on adaptation that introduces adaptable interfaces, interactions, and processing for Linked Data Platform components. We back up the approach with an implementation that enables components to provide adaptation through a thin layer on top of or integrated with their domain data and functionality. Finally, we conduct an evaluation with respect to the DLUBM Linked Data benchmark environment by adapting at runtime the interfaces, interactions, and processing of the involved components in order to execute queries in centralized and decentralized settings.

REFERENCES

[1] Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch, and Richard Cyganiak. 2017. Linking Open Data cloud diagram. (2017). http://lod-cloud.net/

[2] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A survey. *Computer Networks* (2010).

[3] Tim Berners-Lee and Dan Connolly. 2011. *Notation3 (N3): A readable RDF syntax.* Team Submission. W3C. http://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328//. Latest version available at https://www.w3.org/TeamSubmission/n3/.

[4] Christian Bizer, Tom Heath, and Tim Berners-Lee. 2009. Linked Data – The Story So Far. *International Journal on Semantic Web and Information Systems* (2009).

[5] Enrico Daga, Luca Panziera, and Carlos Pedrinaci. 2015. A BASILar Approach for Building Web APIs on top of SPARQL Endpoints. In *Proceedings of the Workshop on Services and Applications over Linked APIs and Data at the European Semantic Web Conference.*

[6] Simon Duquennoy, Gilles Grimaud, and Jean-Jacques Vandewalle. 2009. The Web of Things: interconnecting devices with high usability and performance. In *Proceedings of the International Conference on Embedded Software and Systems.*

[7] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation. University of California, Irvine, USA.

[8] Dominique Guinard, Vlad Trifa, Friedemann Mattern, and Erik Wilde. 2011. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In *Architecting the Internet of Things.* Springer Berlin Heidelberg.

[9] Antonio J. Jara, Alex C. Olivieri, Yann Bocchi, Markus Jung, Wolfgang Kastner, and Antonio F. Skarmeta. 2014. Semantic Web of Things: an analysis of the application semantics for the IoT moving towards the IoT convergence. *International Journal of Web and Grid Services* (2014).

[10] Felix Leif Keppmann, Maria Maleshkova, and Andreas Harth. 2016. Semantic Technologies for Realising Decentralised Applications for the Web of Things. In *International Conference on Engineering of Complex Computer Systems.*

[11] Johannes Koch, Carlos A. Velasco, and Philip Ackermann. 2017. *HTTP Vocabulary in RDF 1.0.* Working Group Note. W3C. https://www.w3.org/TR/2017/NOTE-HTTP-in-RDF10-20170202/. Latest version available at https://www.w3.org/TR/HTTP-in-RDF10/.

[12] Reto Krummenacher, Barry Norton, and Adrian Marte. 2010. Towards Linked Open Services and Processes. In *Proceedings of the Future Internet Symposium.*

[13] Albert Meroño-Peñuela and Rinke Hoekstra. 2016. grlc Makes GitHub Taste Like Linked Data APIs. In *Proceedings of the European Semantic Web Conference [Satellite Events].*

[14] Dennis Pfisterer, Kay Romer, Daniel Bimschas, Oliver Kleine, Richard Mietz, Cuong Truong, Henning Hasemann, Alexander Kröller, Max Pagel, Manfred Hauswirth, Marcel Karnstedt, Myriam Leggieri, Alexandre Passant, and Ray Richardson. 2011. SPITFIRE: Toward a Semantic Web of Things. *IEEE Communications Magazine* (2011).

[15] Kristina Sahlmann and Thomas Schwotzer. 2015. MOCAP: Towards the Semantic Web of Things. In *Proceedings of the International Conference on Semantic Systems [Posters and Demos].*

[16] Steve Speicher, John Arwe, and Ashok Malhotra. 2015. *Linked Data Platform 1.0.* Recommendation. W3C. http://www.w3.org/TR/2015/REC-ldp-20150226/. Latest version available at http://www.w3.org/TR/ldp/.

[17] Sebastian Speiser and Andreas Harth. 2011. Integrating Linked Data and Services with Linked Data Services. In *Proceedings of the Extended Semantic Web Conference.*

[18] Steffen Stadtmüller. 2016. *Dynamic Interaction and Manipulation of Web Resources.* Ph.D. Dissertation. Karlsruhe Institute of Technology.

[19] Steffen Stadtmüller, Sebastian Speiser, and Andreas Harth. 2013. Future Challenges for Linked APIs. In *Proceedings of the Workshop on Services and Applications over Linked APIs and Data at the European Semantic Web Conference.*

[20] Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, and Rudi Studer. 2013. Data-Fu: A Language and an Interpreter for Interaction with Read/Write Linked Data. In *Proceedings of the International World Wide Web Conference.*

[21] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. 2014. Querying Datasets on the Web with High Availability. In *Proceedings of the International Semantic Web Conference.*

[22] Ruben Verborgh, Thomas Steiner, Davy van Deursen, Rik van de Walle, and Joaquim Gabarró Vallès. 2011. Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In *Proceedings of the International Conference on Next Generation Web Services Practices.*

[23] Jim Webber, Savas Parastatidis, and Ian Robinson. 2010. *REST in Practice: Hypermedia and Systems Architecture.* O'Reilly Media, Inc.