Building REST APIs for the Robot Operating System – Mapping Concepts and Interaction

Felix Leif Keppmann, Maria Maleshkova, and Andreas Harth

Karlsruhe Institute of Technology (KIT), Germany
{felix.keppmann, maria.maleshkova, andreas.harth}@kit.edu

Abstract. The vision of the Web of Things (WoT) aims to leverage Web standards in order to interconnect all types of embedded devices and real-world objects, and thus to make them a part of the WWW. Therefore, WoT aims to build a future Web of devices that is truly open, flexible, and scalable. We aim to contribute towards achieving this goal by relying on existing and well-known Web standards used in the programmable Web (e.g., URI, and HTTP) and the semantic Web (e.g., RDF), in order to enable the Web integration of Robot Operating System (ROS) devices. In particular, we motivate the problem of integrating ROS devices in Web environments, elaborate on the integration potential, and describe specific application examples. We provide a mapping between ROS and REST concepts and interaction primitives. In addition, we show how REST is capable to enhance a mapping of the ROS architecture in terms of complex resources and hypermedia. The contributions described in this paper pave the way towards realising a WoT, where ROS devices can be easily accessed and directly integrated by using standard Web technologies, without additional custom implementation effort or having to add intermediate communication layers.

1 Introduction

Current developments of the Web are characterised by the wider integration of network-enabled devices, which serve as data providers or actuators, in the context of client Web applications. However, even though real-life objects can finally participate in Web scenarios, the use of individual and specific interaction mechanisms and data models lead to realising isolated islands of connected devices or to custom, not reusable solutions. Devices are increasingly network-enabled but rely on heterogeneous network communication mechanisms, use no standardised interfaces and introduce new data models for each individual device. This results in small groups of Web-enabled interconnected devices, which cannot be directly extended and reused for different application domains or different scenarios but are instead network and domain specific.

The bridging of these isolated groups of connected devices and the provisioning of overall interoperability are required in order to enable a pervasive Web of Things (WoT). The vision of the WoT is to leverage Web standards in order to interconnect all types of embedded devices (e.g., sensors, mobile

phones, etc.) and real-world objects, and thus to make them a part of the World Wide Web (WWW). Therefore, WoT aims to build a future Web of devices that is truly open, flexible, and scalable. We aim to contribute towards achieving this goal by relying on existing and well-known Web technologies used in the programmable Web (e.g., Uniform Resource Identifier (URI), and Hypertext Transfer Protocol (HTTP)) and the semantic Web (e.g., Resource Description Framework (RDF)). In particular, we focus on the integration of the Robot Operating System (ROS) [5] as a specialised architecture for robotic systems and focus on exploring the synergies emerging from the combination of ROS and Representational State Transfer (REST) [1]. We develop a solution based on REST, proposed as an architectural style for integrating loosely coupled distributed systems. We map ROS, which is an established standard and framework in the robotics area, to the concepts and interaction mechanisms of REST. As a result, devices participating in robotic systems can be accessed in a uniform way – directly over HTTP, using URIs, and adhering to the REST architectural style. In summary, we make the following contributions:

- We provide a conceptual mapping between ROS and REST.
- We describe how to map the interaction mechanism between ROS and REST.
- We show how hypermedia may enhance a distributed ROS application.

The contributions described in this paper pave the way towards realising a WoT, where ROS devices can be easily accessed, deployed and integrated directly by using standard Web technologies, without additional custom implementation effort or having to add intermediate communication layers.

The remainder of the paper is structured as follows: in Section 2 we further motivate the problem, elaborate on the integration potential and describe specific application examples. In Section 3 we provide an overview of the building blocks of the ROS architecture. In Section 4, we first develop a generic view on resource-oriented architectures and then create a mapping between the concepts and interaction mechanisms of REST and ROS. We provide a short overview of related work in Section 5 and conclude the paper in Section 6.

2 Motivation

The vision of a pervasive Web, e.g., promoted by the Internet of Things (IoT), requires the integration of heterogeneous software and hardware. Whenever it comes to making architectural decisions, there is a trade off between the general applicability of the architecture and the development of specialised architectures that may be more efficient for specific use cases. We advocate an approach that supports the development of specialised solutions by enabling their subsequent integration in a broader context.

We illustrate this challenge with the following example. In the field of robotics the ROS architecture is a key player for the efficient integration of modularised robotic systems. These robotic systems are increasingly used in a broader context, for instance for developing flight and cockpit simulators, which are characterised by the use of specialised hardware and software elements. While the

individual hardware and software components would require custom implementations and solutions, ROS enables an overall unified communication and handling of the simulator systems. The result is an isolated group of interconnected devices (flight and cockpit simulator components) that can communicate in a unified way, since they conform to ROS. This specialised solution can be extended with REST-based communication in order to enable its Web integration, thus facilitating its adaptability within the WoT. This example emphasises that pervasive integration scenarios, as promoted by the IoT and WoT vision, can benefit from specialised architectures (i.e., ROS) for specific use cases in combination with web-scale architectures (i.e., adhering to the REST architectural style), which are capable of overcoming the heterogeneity.

We take the example of a ROS flight and cockpit simulator as a specific scenario used for illustrating the challenges faced while developing an approach for the mapping between the used ROS architecture and a resource-oriented architecture based on REST and Linked Data. In particular, the challenge of integrating these two architectural approaches is two-fold. First, the integration has to be realised on a conceptual and interaction level. With REST, we introduce a resource-oriented viewpoint on distributed applications. The basic concepts associated with REST and the corresponding implications on the interactions within the distributed application must be aligned with the particular ROS architecture. Second, the integration has to be realised on the semantic level, based on the meaning of the processed data, thus facilitating the unified handling of heterogeneous data formats and data sources. By introducing RDF as way for formally specifying data models and Linked Data for publishing and interlinking data, we provide the foundation for the integration of data having different formats and coming from different sources. In the following, we focus on the first challenge – providing a mapping on the conceptual and interaction level. This work serves as a starting point for the unified handling of ROS, by adopting REST-based communication.

3 Robot Operating System

We provide an overview of important concepts, interaction mechanisms, data models and formats of ROS. ROS provides a coherent way for identification and message transmission in a distributed application. A distributed application is composed of ROS *nodes*, which communicate via *messages* with other nodes by using ROS *topics* and ROS *services*, or by utilising ROS *parameters*.

ROS introduces the following ***Concepts***:

Name ROS names enable the identification of every component in a distributed ROS system via a central hierarchical naming scheme. The naming is based on a slash-separated identifier for each resource in the ROS application, e.g., /ns/node. Names start with an alpha character, including forward slash and tilde, followed by alphanumeric characters, including forward slashes and underscores. The global namespace is identified by a forward slash and subsequent slashes separate different namespaces within an identifier. Resources

may access other resources in or above their own namespace but only create resources in their own namespace.

Node ROS nodes are the basic building blocks of the architecture and perform computation in a ROS application. A ROS system may consist of several interconnected nodes each handling the computation of a relatively narrow functionality, e.g., location tracking, or laser sensor operation. A node is uniquely identified by a name and all nodes in an application communicate via topics, services, or the parameter server. Each node has a node type, encapsulates its functionality and provides a minimal Application Programming Interface (API) which is exposed to the rest of the nodes. The type of a node defines on file system level within the package the name of the executable to be executed when the node is accessed.

Master The ROS master provides a centralised name system and registration facility for nodes, as well as for their published topics and services in the ROS system. It enables nodes to discover and locate other nodes, and tracks all subscribers and publishers of topics and services. The communication of messages between nodes over topics is, similar to services, delegated to the nodes and not handled by the master. The functionality of the master is accessible through an API based on Extensible Markup Language Remote Procedure Call (XML-RPC).

ROS introduces the following ***Data*** elements:

Message ROS messages are simple data structures of typed fields, which are exchanged in the communication between nodes in a ROS resource graph. A message is typed by a message type, which defines the structure of the contained data. ROS packages may define these message types in simple text files based on a set of build-in field types. Included in each message is the version of the message type, which is based on the Message-Digest Algorithm 5 (MD5) hash of the underlying message type file. Only nodes with the same version, i.e., MD5 hash, are allowed to communicate messages of this particular message type.

Bag ROS bags are a collections of serialised messages for persistent storage and later reuse, e.g., playback of messages. The original representation used by the ROS transport layer is utilised by bags as data format, which leads to efficient processing or replaying of messages.

Finally, ROS introduces the following ***Interaction*** primitives:

Topic ROS topics provide an anonymous publish-subscribe mechanism for the interaction in a ROS system and enable unidirectional distribution of messages from a specific node to a number of interested nodes. A topic is identified by a name and is strongly typed for one kind of messages, i.e., the type of a topic is the same as the type of the messages to be distributed by the topic. Nodes receive messages of this message type only if they subscribed beforehand to the topic.

Service ROS services provide a request-response mechanism for the interaction in a ROS system. A service is identified by a name and has a message pair,

```
Topics
        /aircraft/engine/speed          > 0 - 1000
Services
        /aircraft/start
        /aircraft/stop
        /aircraft/set_power             < on | off
        /aircraft/get_power             > on | off
        /aircraft/engine/set_speed      < 0 - 1000
        /aircraft/engine/set_direction  < 0 - 360
        /aircraft/engine/get_direction  > 0 - 360
```

Listing 1. Aircraft Topics/Services in ROS

i.e., a request message and a response message. Similarly to topics, each service is strongly typed, based on and versioned by a MD5 hash of the service file, which includes, in contrast to topics, the types of both messages. The interaction with a service is synchronous, i.e., a node sends a request message to a service and waits for the response message.

Parameter The ROS server provides at runtime a shared dictionary for parameter storage in nodes in the resource graph. The server is not designed for high performance use cases but for rather static and low volume data, e.g., configurations. The identification of parameters follows the ROS name scheme. Single or tree-based access to the shared parameter storage is granted trough an API based on XML-RPC.

4 Mapping

In this section we provide a mapping between a distributed application architecture based on the technologies proposed by REST and the ROS architectures. We 1) map the concepts between both architectures; 2) map the interaction mechanisms utilized in the communication; 3) indicate how the REST architectural style enables modelling of structural information and hypermedia, which is implicitly expressed in ROS.

In Listing 1 we show an abstract aircraft modelled in ROS as a topic and a number of services, which will serve as an example in the following. Each line represents a topic or service identified by a name with input or output values. The start/stop as well as the set/get power services start and stop the engine. Via the set speed service a new speed may be set and the get/set direction service is responsible for the direction in terms of angular degree. The topic informs a subscriber about the changed speed of the aircraft.

4.1 Concepts

The REST architectural style proposes the use of common Web technologies, e.g., URIs, and HTTP. In Table 1 we provide a mapping between these concepts and

Concept	HTTP	HTTP/RDF	ROS
Node	Host	Host	Node
Dist. App.	Network of Hosts	Network of Hosts	Network of Nodes
Resource	HTTP Resource	HTTP Resource	Service, Parameter, Topic Subscriber
Representation	XML, JSON, ...	Turtle, JSON-LD, ...	Message, XML
Data Model	-	RDF	Message Format, XML-RPC Schema
Identifier	URI	URI	Name
Transport	HTTP	HTTP	TCPROS, UDPROS
Interaction	HTTP Verbs	HTTP Verbs	Methods provided by Topic, Service, and Parameter

Table 1. Concept Mapping between HTTP, HTTP/RDF and ROS

concepts of the ROS architecture and we have also added RDF as data model. For a HTTP-based architecture, we are able to identify the basic concepts, i.e., node, distributed application, and resource. Any component capable of providing resources via HTTP at a DNS name or an IP address may serve as node. A network of several communicating nodes composes a distributed application, which provides a higher-value functionality. HTTP resources expose their state, and thereby a partial state of the node, to the network. We are obliged to use specific concepts for identification and transport. In particular, HTTP provides also the underlying protocol for the transport of messages and, as described in Section 4.2, for the interaction between nodes. Based on HTTP, we use URIs to uniquely identify resources. We are not obliged to adhere to a pre-described data model and representation format, since neither HTTP nor the REST architectural style include a particular model. Any data format is permitted for serialising a representation of a state, although some specific data formats are frequently used, e.g., JSON, or XML. For the differentiation of representation formats we may use mime types.

In addition to the REST style, we asses the use of Linked Data techniques, which we aim to use in future work to extend the mapping with a flexible semantically powerful data model and respective data formats. As shown in Table 1, we close the gap by introducing RDF for defining a common data model. Several data formats exist for the serialisation of representations adhering to the RDF model, e.g., Turtle, JSON-LD, RDF/XML, N-Triples, or N3.

Similarly to HTTP, we are able to identify the basic concepts in a ROS architecture. A distributed application based on ROS is a network of ROS nodes, which interact in order to provide a higher-value functionality. The state of a ROS node is exposes by services, parameters, or indirectly by the subscriber of a topic. Therefore, different types of resources exist in the ROS architecture. In contrast to HTTP, ROS provides data models and formats for representation. The message format provides a simple model for messages exchanged via topics and services, serialised on transport level in a ROS-specific data format. Repre-

Architecture	Resource Type	CREATE	READ	UPDATE	DELETE
HTTP	HTTP Resource	POST/PUT	GET	PUT/PATCH	DELETE
ROS	Service	–	get*	set*	–
ROS	Topic Subscriber	–	–	publish	–
ROS	Parameter	setParam	getParam	setParam	deleteParam

Table 2. Interaction Mapping between HTTP and ROS

sentations of parameters adhere to the XML-RPC schema and are serialised as XML. The ROS architecture provides a global naming scheme for services, topics, and parameters, i.e., ROS names are unique identifiers for resources. With TCPROS and UDPROS the architecture provides two implementations of the abstract ROS transport protocol for the transfer of messages between nodes. Negotiation procedures transparently handle the optimal choice of the protocol implementation, based on preferences of the involved nodes, and enable the extensions with further protocols.

With the concept of resources exposing parts of a system to a network and unique identifiers to identify these resources we allow a mapping which is categorized as level one in Richardson's maturity model [6] for REST services.

4.2 Interaction

Derived from the underlying HTTP proposed by the REST style, which enables transport and interaction, we can use the full range of CRUD operations on resources. The HTTP verbs POST, GET, PUT, and DELETE map to the CRUD operations in the interaction with resources, as shown in Table 2. Two generic types of resources – atomic resource and collection resource, react with a different behaviour on these methods. Atomic resources are created or updated by executing PUT on a URI, read by GET and deleted by DELETE. A collection resource creates a new resource in the collection after a POST request is executed on the URI. The resource is read by executing GET, providing by convention a list of contained resources, and is deleted by DELETE. In addition, the PATCH verb was introduced to allow, in contrast to PUT, partial updating of resources.

We are able to identify different interaction mechanism in the ROS architecture, depending on the type of resource, i.e., specific interaction mechanisms exist for topics, services, and parameters. Depending on the type of resource, we partially map the mechanisms to the generic CRUD operations, shown in Table 2. ROS topic subscribers provide, on a conceptional level, only one atomic resource, which can be updated by a topic publisher as a single, and thus distinct, CRUD operation. Creation and deletion of resources in a collection resource of the subscriber is not supported, i.e., the subscriber provides only one resource for updates by a publisher. The resource does not support a read by other systems, but a node may expose the state again as topic or service. ROS services do not provide distinct CRUD operations. In particular, the creation and deletion of services is not possible at runtime, i.e., a service is an atomic pre-defined resource. Furthermore, a service defines a set of input and output ROS messages as RPC with custom processing between input and output message. While

```
/aircraft/power                  : on | off
/aircraft/engine/speed           : 0 - 1000
/aircraft/engine/direction       : 0 - 360
```

Listing 2. Aircraft CRUD Service in HTTP

the creation and deletion of a service resource is not possible, the read and update operation may be mapped automatically by introducing getter and setter services. ROS parameter is the only resource type of the ROS architecture supporting all CRUD operations. The master node provides central access to all parameters and interaction mechanism for these parameters. In fact, interaction with parameters is handled by XML-RPC but in a resource-oriented manner. A parameter may be created or updated by the "setParam" method, read by the "getParam" method and deleted by the "deleteParam".

In Listing 2 we show how the abstract aircraft modelled in ROS in Listing 1 is mapped to a HTTP CRUD service. For simplification, we do not use full URIs in the listing but only the path part for identification. By introducing the convention to prefix getter and setter services in names, we automatically map power and direction to HTTP resources. The speed topic is combined with the speed set services to a read-write resource. We may establish a manual custom mapping of the start and stop service.

With the mapping of a constraint set of interaction mechanisms we allow a mapping, which is categorized as level two in Richardson's maturity model for REST services. We point out that the ROS architecture does not provide means for creation and deletion of topics and services at runtime. Only a parameter is mapped completely as a CRUD resources to HTTP. A topic is be mapped to a read-only HTTP resource, which only allows GET. Services allow an automatic mapping to a read-write HTTP resource by introducing getter and setter services.

4.3 Complex Resources & Hypermedia

In the HTTP architecture proposed by REST the granularity of resources is a decision at design time, since there are no restrictions in terms of representation formats. In contrast, the ROS architecture only permits representations composed of basic data types, which leads to relatively fine-grained resources. As a consequence, a resource, which is modelled in REST as one HTTP resource, may be split over several fine-grained topics or services in ROS, i.e., names may contain structural information about resources. In addition, the REST architectural style proposes as a tenet the use of hypermedia to drive the state of an applications, i.e., links connect resources and explicitly relate states of the application.

In Listing 3 we show how the CRUD service of the abstract aircraft in Listing 2 is extended to overcome these shortcomings. For simplification, we do not use full URIs in the listing but only the path part for identification. First, we

```
/aircraft
  linkrel       : start    -> /aircraft
  linkrel       : stop     -> /aircraft
  linkrel       : engine   -> /aircraft/engine
  power         : on | off
/aircraft/engine
  linkrel       : aircraft -> /aircraft
  speed         : 0 - 1000
  direction     : 0 - 360
```

Listing 3. Aircraft Hypermedia Service in HTTP

aggregate fine-grained ROS resources as higher-level HTTP resources, i.e., as the resources aircraft and aircraft engine. Structural information, which was encoded in ROS names, is now obsolete. Second, we expose the application state, which is implicitly present in our ROS example, explicit by utilizing hypermedia, i.e., by relating the resources with typed links.

Links, which drive the application state are noted as `linkrel` in the listing. The link `start` appears in the representation of an aircraft as long as the aircraft is not started. The link `engine` appears as long as the aircraft is started. The link `stop` appears as long as the aircraft is started and the speed of the engine is zero. An engine resource exists as long as the aircraft is started and its representation contains a link back to the aircraft. A client, capable to interpret the link types, is able to use the aircraft system in a correct way by accessing a single URI as starting point, in this case the aircraft.

The ROS architecture does not provide means for creating complex resources and interlinking these resources. By aggregating fine-granular ROS resources to higher-level HTTP resources and by adding hypermedia, we enhance the mapping and expose a ROS system as proposed by the REST architectural style, i.e., the service now adhere to level three of Richardson's maturity model. However, the mapping of the application semantics is currently a manual design decision. To what extent we can (semi-)automate the mapping of semantics is part of future work.

5 Related Work

In the context of IoT, [2,3] propose the use of established Web technologies, i.e., the integration of smart things following the REST architectural style. The main target of IoT is the connection and combination of various network-enabled devices and virtual artefacts – smart things – to distributed applications. This integration is primarily based on network connectivity between the objects but may lead to incompatible digital islands, i.e., distributed applications in IoT that are closed environments. By opening these digital islands via common web technologies, i.e., REST, the authors introduce the WoT vision. The WoT approach supports our viewpoint on distributed applications consisting of a network of

relatively independent nodes and the data flow between these nodes, enabled by push or pull interaction.

The underlying design goals of ROS are described in [5]. They elaborate on the thin peer-to-peer messaging layer of ROS with an own language-neutral Interface Definition Language (IDL), the message types, and code generators for all supported programming languages. With a number of specific use cases, the authors show the intended purpose of ROS and relate it to other robotic systems. A cross-domain use case of ROS in the surgical robotics research is described in [4]. This example shows in more detail how ROS, as specialised architecture, efficiently solves suitable use cases, while in the broader context more generalised architectures may serve better to solve the integration of heterogeneous systems.

6 Conclusion

Recent developments on the Web are characterised by the growing use of sensors and embedded devices, which have an increasing importance in the contest of building high-value user applications but also more complex distributed solutions. This trend raises new issues around the question of how can devices and 'things' be seamlessly integrated and become an integral part of the Web. To this end, in the paper we focus on robot systems and elaborate on the challenges related to integrating ROS devices in Web environments. We provide a solution in the form of a mapping between ROS and REST concepts and interaction primitives. Furthermore, we show how REST is capable to enhance a mapping of the ROS architecture in terms of complex resources and hypermedia. The contributions described in this paper pave the way towards realising a WoT, where ROS devices can be easily accessed and directly integrated by using standard Web technologies, without additional custom implementation effort or having to add intermediate communication layers.

References

1. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine, USA (2000)
2. Guinard, D., Trifa, V., Mattern, F., Wilde, E.: 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 (2011)
3. Guinard, D., Trifa, V., Wilde, E.: A Resource Oriented Architecture for the Web of Things. In: Proceedings of the Internet of Things Conference (2010)
4. Hannaford, B., Rosen, J., Friedman, D.W., King, H., Roan, P., Cheng, L., Glozman, D., Ma, J., Kosari, S.N., White, L.: Raven-II: An Open Platform for Surgical Robotics Research. IEEE Transactions on Biomedical Engineering 60(4), 954–959 (Apr 2013)
5. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System. In: Proceedings of the ICRA Workshop on Open Source Software. vol. 3, p. 5 (2009)
6. Webber, J., Parastatidis, S., Robinson, I.: REST in Practice: Hypermedia and Systems Architecture. O'Reilly (2010)