Give Agents Some REST: A Resource-Oriented Abstraction Layer for Internet-scale Agent Environments

Andrei Ciortea[1], Olivier Boissier[1], Antoine Zimmermann[1], and Adina Magda Florea[2]

[1] Univ. Lyon, MINES Saint-Étienne, CNRS,
Lab Hubert Curien UMR 5516, F-42023 Saint-Étienne, France
{andrei.ciortea,olivier.boissier,antoine.zimmermann}@emse.fr
[2] Department of Computer Science, University "Politehnica" of Bucharest,
313 Splaiul Independentei, 060042 Bucharest, Romania
adina.florea@cs.pub.ro

Abstract. To keep up with current technological developments, the engineering of multi-agent systems (MAS) has to provide solutions to: (i) support large scale systems, (ii) cope with open systems, and (iii) support humans in the loop. In this paper, we claim that the World Wide Web provides a suitable middleware for engineering MAS that address these challenges in an integrated manner. Even though approaches to engineer Web-based MAS have already been explored in the MAS community, existing proposals do not achieve a complete integration with the Web architecture. We approach this problem from a new angle: we design the *agent environment* as a *hypermedia application*. We apply REST, the architectural style of the Web, to introduce a resource-oriented abstraction layer for agent environments that decouples the *application environment* from its *deployment context*. Higher-level environment abstractions can then be implemented on top of this lower-level abstraction layer. To demonstrate our approach, we implemented a multi-agent application for the Internet of Things in which software agents can seamlessly navigate, use and cooperate in an environment deployed over multiple Web services (e.g., Facebook, Twitter) and constrained devices.

Keywords: Agent environments, service-oriented computing, REST, Web architecture, Web of Things, Internet of Things

1 Introduction

Over the last decade, the agent environment has gained broad recognition as a *first-class abstraction* in multi-agent systems (MAS) [39]: it is a key component designed and programmed with clear-cut responsibilities, such as mediating interaction among agents and access to the deployment context (e.g., physical devices, digital services). The increased emphasis on agent environments in MAS also raises new challenges to be addressed. Three research topics of growing

importance are to design agent environments that [38]: (i) support *large scale systems*, (ii) can cope with *open systems* in which components are deployed and evolve at runtime, and (iii) *support humans in the loop*.

If we are to examine existing software systems based on the above criteria, the World Wide Web is arguably the most scalable, versatile, and human-centric software system deployed on the Internet. In fact, the Web was specifically designed to be an Internet-scale and long-lived system in which components can be deployed and can evolve independently from one another at runtime [13]. More recently, on account of its architectural properties, the Web is emerging as the application layer for the Internet of Things (IoT), an initiative known as the Web of Things (WoT) [23,40]: physical devices are integrated into the Web such that software clients can access them in a uniform manner using Web standards. The WoT vision is rapidly being implemented through combined standardization efforts of the W3C WoT Working Group[3], the IETF Constrained RESTful Environments Working Group[4], and the IRTF Thing-to-Thing Research Group[5]. The World Wide Web is turning into a middleware for most systems envisioned on the Internet, and its huge success comes from its carefully designed architectural properties.

In this paper, we take a deep look into the rationale behind the modern Web architecture. Our claim is that we can apply the same rationale to address the three above-mentioned research topics in an integrated manner. The novelty of our approach is to design the *agent environment* as a *hypermedia application*. We apply REST [14], the architectural style of the Web, to introduce a resource-oriented abstraction layer for agent environments that decouples the *application environment*[6] from its *deployment context*. This abstraction layer is based on *socio-technical networks (STNs)* [2], that is dynamic networks of humans, software agents and artifacts interrelated in a meaningful manner via typed relations. We use STNs to address HATEOAS, one of the architectural constraints that is really central to REST and the modern Web architecture.[7] To the best knowledge of the authors, this proposal is the first approach to engineer MAS that are completely aligned with the Web architecture.

This paper is structured as follows. Section 2 presents the REST architectural style and discusses related work on engineering Web-based MAS. Section 3 presents our approach in further detail. Section 4 presents an implementation of this approach to develop a multi-agent application for the IoT in which software agents can seamlessly navigate and use an agent environment deployed over Facebook, Twitter, and multiple constrained devices.

[3] `http://www.w3.org/WoT/WG/`, last accessed: 08.02.2017.
[4] `http://datatracker.ietf.org/wg/core/`, last accessed: 08.02.2017.
[5] `http://datatracker.ietf.org/rg/t2trg/`, last accessed: 08.02.2017.
[6] That is to say, the part of the environment designed and programmed for the application at hand [39].
[7] In previous publications, STNs have been applied as a means to bring the multi-agent paradigm to IoT application development [4], and to enhance discoverability across otherwise siloed IoT environments [3].

2 Background and Related Work

In Section 2.1, we present the key principles behind REST and the architectural properties they induce. Then, in Section 2.2 we focus our attention on existing approaches to engineer Web-based MAS. We conclude that none of the existing approaches are completely aligned with the REST principles.

2.1 The REST Architectural Style

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems [14]. REST was designed to meet the needs of the Web as an *Internet-scale system* in which the overall performance is dominated by network communication rather than computation-intensive tasks [13]: component interactions consist of large-grain data objects transferred over high-latency networks that span across geographical and organizational boundaries. To this end, REST was designed to minimize latency and network communication, and at the same time maximize the independence of components and the scalability and visibility of component interactions [14,13]. These combined characteristics transformed the Web into an *Internet-scale*, *open*, and *long-lived* system.

REST in a Nutshell REST is defined as a *coordinated set of architectural constraints* [14]. When applied to a software architecture, REST constrains only those portions of the architecture considered to be essential for *Internet-scalability* and *openness* [13]. In particular, REST focuses on the *semantics of component interactions*, in contrast to other architectural styles that focus on the *semantics of components*. REST can thus be used in conjunction with component-based specifications, such as the FIPA Abstract Architecture [15] or the reference model for agent environments proposed in [39].

Interaction in a REST-style system follows the *request-response* pattern: a *client* (e.g., a Web browser) issues a request, a *server* processes the request and returns a response. REST components (i.e., origin clients, intermediaries, origin servers) interact by transferring *representations* of *resources*. A *resource* is the key abstraction of information in REST, where "any information that can be named is a resource" [14]. The interaction is *stateless*, which improves the scalability of server implementations, and performed through a *uniform interface* that hides the heterogeneity of component implementations. When combined, the *stateless interaction* and *uniform interface* constraints enable the use of *intermediaries* that can interpret requests almost as well as their intended recipients. Intermediaries can then be deployed along the request-response path, for instance, to cache and reuse interactions, to distribute the workload across multiple servers, to enforce security or to encapsulate legacy systems [14], which are all important concerns in an *Internet-scale*, *open*, and *long-lived* system.

Hypermedia-driven Interaction To achieve a *uniform interface* between components, interaction in REST is *driven by hypermedia*, a constraint a.k.a.

Hypermedia As the Engine of Application State (HATEOAS) [14]. Without hypermedia-driven interaction, browsers would be tightly coupled to origin servers and unable to seamlessly navigate an *open* Web.

To best illustrate hypermedia-driven interaction, consider a typical Web application composed of multiple hyperlinked Web pages. This application is a finite state machine: each page represents a state and hyperlinks between pages represent transitions between states. Given the URI of one of the pages, a client can dereference the URI to retrieve an *HTML representation* of that page. This action thus triggers a transition to a new application state, and if the transition is completed successfully the client can now choose from a new set of Web pages (i.e., new reachable states). Both the *next reachable states* and the *knowledge required* to transition to those states are conveyed to the client *through hypermedia* retrieved from the origin server (e.g., via HTML forms).

In other words, given a URI as an *entry point* into a Web application, *standard Web transfer protocols* and *representation formats* (a.k.a. *media types*), the client should be able to discover new resources and how to use those resources at runtime. The client is thus loosely coupled to the origin server via standardized knowledge: it does not hard-code URIs or individual requests. This is an important difference from how most existing Web services work today.

HATEOAS is essential to achieve a uniform interface, which in turn is a central feature in REST [14]. The uniform interface allows components to be deployed and to evolve independently from one another, which is an important feature in an *Internet-scale*, *open*, and *long-lived* system.

2.2 Engineering Web-based MAS

In this section, we discuss approaches to use the Web as an infrastructure for distributed MAS. These approaches fall broadly in one of two categories: they use the Web either as a *transport layer*, or as an *application layer*. To the best knowledge of the authors, none of the existing approaches fully complies with the REST principles presented in the previous section.

The Web as a Transport Layer Systems that use the Web as a transport layer make limited use of its architectural properties, existing infrastructure and future extensions (see Section 6.5.3 in [14] for a detailed discussion).

The *Foundation for Intelligent Physical Agents (FIPA)* investigated the integration of software agents with Web services [21]. One of the main Web-related results stemming out of the standardization efforts is a FIPA specification for using HTTP as a transport protocol for messages exchanged among agents [16]. FIPA-compliant platforms that implement this specification include JADE[12], SPADE [22] and SEAGENT [11].

WS- Web services* also use the Web as a transport layer [32]. Given their widespread adoption in the early 2000s, the WS-* standards have had a strong influence on multi-agent research in service-oriented computing [25,37,24,18]. A number of MAS platforms support direct integration with WS-* Web services

(e.g., [33,5,11]), while others rely on a gateway component to mediate interactions between agent services and Web services (e.g., [20,31,36,29]).

The Web as an Application Layer More recent approaches to engineer Web-based MAS are based on REST-like Web services (commonly referred to as "RESTful"), which typically use HTTP as an *application protocol* (see [32]), but do not use hypermedia and thus do not support hypermedia-driven interaction. As a result, clients and origin servers are *tightly coupled* to one another, which is an important limitation when engineering Internet-scale, open systems.

In [9], the authors propose an approach to automatically translate agent services to Web services (and vice-versa), with the aim to support both the WS-* standards and REST. The generated REST-like Web services, however, are not resource-oriented and do not support hypermedia-driven interaction. In other proposals, REST-like Web services are used for agent communication. In [1], agents implement interaction protocols (e.g., FIPA Contract Net Protocol [17]) by creating and manipulating resources on a Web server. A more general approach to REST-inspired agent communication is taken in [35]: agents use *message repositories* to create *graphs of messages* following predefined interaction protocols. Note that the links among messages can help address the HATEOAS constraint, but details on the APIs exposed by the message repositories are not available (e.g., if they are hypermedia-driven).

Radigost [28] implements the FIPA abstract architecture and exposes most of the system functionality through a REST-like Web service. A gateway component allows agents hosted on other platforms to exchange messages with Radigost agents. However, the HATEOAS constraint is not addressed.

REST-A [19] provides a REST-like approach to design *agent environments*: agents perform actions on *resources* in their environment using a set of CRUD (create, read, update, delete) operations over HTTP. However, the HATEOAS constraint is not addressed.

To conclude, to the best knowledge of the authors, existing approaches to engineer Web-based MAS are either agnostic to REST, or do not support hypermedia-driven interaction. In both cases, this implies that they cannot fully inherit the architectural properties of the Web, such as *Internet-scalability* and *openness*.

3 A Resource-oriented Foundation for Agent Environments

In this section, we propose a resource-oriented abstraction layer for agent environments that goes beyond the state-of-the-art by fully conforming to REST. As illustrated in Figure 1, the purpose of this layer is to decouple the *application environment* from its *deployment context*. Agents can then interact with their environment while being agnostic to the underlying infrastructure. We present our approach in Section 3.1. Then, in Section 3.2 we focus our discussion on

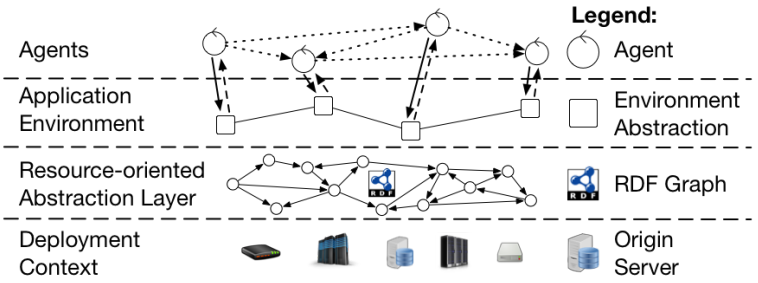

Fig. 1. Layers of an agent environment: the resource-oriented abstraction layer decouples the *application environment* from its *deployment context*.

achieving a *uniform interface* between the programmable part of an agent environment, which would typically run on a MAS platform, and its deployment context (e.g., physical devices, digital services).

3.1 Resource-oriented Agent Environments

Our approach to design the proposed resource-oriented abstraction layer for agent environments relies on *socio-technical networks (STNs)*, which were defined formally in [2]. STNs are grounded in multi-agent research, based on semantic Web technologies, and aligned with the Web architecture.

In the following, we first present the core abstractions introduced by the STN model [2]. Then, we show how they map to Web resources, and finally discuss the development of application environments on top of these abstractions.

Socio-technical Networks *Socio-technical networks (STNs)* are dynamic networks of humans and things (e.g., physical devices, digital services) interrelated in a meaningful manner via *typed relations* (e.g., friendship, ownership, provenance, colocation). STNs are situated in environments that span across the physical-digital space. Humans and things that are actively trying to influence the environment are modeled as *agents*, and things that passively augment the environment with new capabilities as *artifacts*. Agents can enter or leave STNs and "rewire" their networks in pursuit of their goals.

The *agent* and *artifact* abstractions have their roots in the *Agents & Artifacts meta-model* [30] and are motivated by the separation of concerns principle. First, these abstractions separate exhibited behavior from the actual entities, which allows developers and end-users to conceive of humans and heterogeneous things in a uniform manner. Second, agents and artifacts simplify the design of multi-agent applications by providing a clear separation between the logic that manipulates the environment and the logic that augments the environment. Third, agents and artifacts provide a modular approach to multi-agent application development when designed and implemented as loosely coupled components meant to be deployed and to evolve independently from one another at runtime.

Typed relations enable the *dynamic discovery* of agents and artifacts at runtime. For instance, say David owns a smart TV that can aggregate movie recommendations from other smart TVs owned by David's friends. If David becomes friends with Bob, an explicit representation of this relation (e.g., via Facebook or Twitter) enables David's smart TV to discover Bob and any smart TVs he might own.

From STNs to Web Resources An STN is reflected in the digital world by means of *digital artifacts* hosted by *STN platforms*. For instance, a lightbulb is a physical artifact that can have a digital counterpart hosted by an STN platform. The *digital lightbulb* represents the *physical lightbulb* and reflects its state. A software agent can then turn on or off the physical lightbulb by updating the state of the digital lightbulb. Similarly, if a human agent turns on or off the physical lightbulb, its state is reflected by the digital lightbulb and can be percevied by software agents.

Human and software agents can also be represented by digital artifacts, such as the *user accounts* they may hold on various STN platforms. User accounts are the agents' proxies in STNs: it is assumed that the entity acting through a user account is acting on behalf of the agent who holds the account.

To deploy STNs on the Web, we map all entities in an STN (e.g., agents, physical artifacts, digital artifacts, STN platforms) to *Web resources* described in the *Resource Description Framework (RDF)* [10]. RDF is the data model of the Semantic Web and a natural choice for our purposes. We provide a Web ontology[8], called hereafter the *STN ontology*, that developers can use to describe the various entities and relations among them in conformance with the proposed STN model [2]. Developers can further extend the STN ontology with domain- and application-specific knowledge.

STN-based Application Environments A key feature that STNs bring to agent environments is the *dynamic discovery* of agents and artifacts via crawling. This feature is central to *Internet-scale* and *open* MAS that can evolve over time. The state of a MAS is reflected in the resource-oriented abstraction layer, for instance whenever an agent joins or leaves the system, or whenever an artifact is created or deleted. Agents can navigate and act on the STN-based layer via environment abstractions (see Figure 1), which would typically run on a MAS platform. For MAS platforms that do not support environment abstractions, the manipulation of the STN-based layer can simply be hidden behind the agents.

Note that crawling STNs at scale would be inefficient, but STN-based application environments can provide agents with more efficient discovery mechanisms built on top of the proposed abstraction layer. By analogy, the Web enables the discovery of Web pages, but manually browsing the Web to search for a specific Web page is inefficient. Most people would generally start with a search engine to focus their search.

[8] `http://w3id.org/stn/`

Another point worth emphasizing is that an STN-based application environment is not limited to its underlying STNs or even the Web: it can contain entities that are not reflected in its underlying STNs, and it can integrate non-Web components. For instance, the application environment we present in Section 4 includes an artifact that connects to a cloud-based MQTT broker[9] in order to retrieve sensor readings from a Texas Instruments SensorTag[10]. The MQTT-based artifact is hidden behind the agent using it and not represented in the STN.

3.2 A Uniform Interface for STN-based Agent Environments

To decouple the application environment from its deployment context, it is necessary to achieve a *uniform interface* between the *origin clients* (e.g., MAS platforms, Web browsers) that access and manipulate *digital artifacts*, and the *STN platforms* that host the *digital artifacts*. The "hallmark" of achieving this uniform interface is that the resource-oriented abstraction layer is *driven by hypermedia* (see Section 2.1 for details): given the URI of a Web resource as an *entry point* into this layer, the origin client should then be able to seamlessly navigate and manipulate STNs using standard Web transfer protocols and media types, while being agnostic to the underlying STN platforms.

To achieve this uniform interface, we apply REST's interface constraints[11] to STN platforms. We reformulate these constraints as follows:

- *Identification of digital artifacts*: digital artifacts hosted by an STN platform are always identified via URIs such that they can be referenced globally and independent of context.
- *Manipulation of digital artifacts via representations*: clients interact with STN platforms by exchanging *representations* of digital artifacts, where the representation of a digital artifact is an RDF serialization of the artifact's current or intended state.
- Messages exchanged between clients, STN platforms, and any intermediaries in-between are *self-descriptive*.
- Interaction between clients and STN platforms is *driven by hypermedia*.

We further detail the last two interface constraints in what follows.

Self-descriptive messages In REST, components exchange representations of resources via *self-descriptive messages*: messages are self-contained[12], message semantics are defined by standard *methods* (e.g., the ones defined by HTTP or CoAP) and *representations formats* (a.k.a. *media types*), responses provide

[9] Message Queuing Telemetry Transport (MQTT) is an application-level protocol for the IoT.

[10] `http://www.ti.com/sensortag`, last accessed: 08.02.2017.

[11] See [13] for more details on the interface constraints in REST.

[12] Note that interaction is stateless between requests, so a component does not need to go beyond a single message to understand it.

explicit cache directives etc. Components must be able to reliably interpret exchanged messages using standardized knowledge. Any implementation-specific details about raw resources must remain hidden behind the uniform interface.

When it comes to choosing representation formats for digital artifacts, developers would generally have three options: (i) if applicable, reuse an existing media type[13], (ii) define a new media type, or (iii) use standard RDF serialization formats, such as Turtle or JSON-LD, with ontologies for describing the states of digital artifacts.

The first option, reusing existing media types, promotes interoperability. The second option, defining new media types, can be particularly useful to meet domain-specific requirements, but the new media types have to be adopted to achieve interoperability. For instance, in WoT applications it can be useful to define concise representation formats for lightweight processing on constrained devices. Lastly, using standard RDF serialization formats might be preferable for general purpose applications, but this approach assumes that clients understand the ontologies used to describe the states of digital artifacts.

Hypermedia-driven interaction To support hypermedia-driven interaction, an STN platform must expose a *hypermedia-driven API*. There are multiple solutions already available for this purpose. The STN ontology provides two modules that developers can use to expose hypermedia-driven APIs for STN platforms (see [2] for more details). Other similar solutions include Hydra [27] and the Thing Description[14].

STNs provide several elements to enhance hypermedia-driven interaction both *within* and *across* STN platforms (see [3] for a detailed discussion):

- the *social network metaphor* enables the discovery of agents;
- the `stn:holds`[15] relation, defined in the STN Ontology, enables the discovery of *user accounts* held by *agents* in various STNs;
- the `stn:hostedBy` relation enables the discovery of STN platforms and any hypermedia-driven APIs they may expose.

However, most existing Web services, such as Facebook and Twitter, expose non-hypermedia APIs. Nevertheless, both Facebook and Twitter can bring significant value to STN-based agent environments, as we show in Section 4. To support our approach, we thus have to provide solutions to integrate non-hypermedia APIs into the hypermedia-driven STN-based layer. We addressed this problem in [3].

To conclude, in this section we introduced a resource-oriented abstraction layer for agent environments. This layer is based on STNs, which are used to

[13] A list with all registered media types is available at IANA: `http://www.iana.org/assignments/media-types/media-types.xhtml`, last accessed: 08.02.2017.

[14] `http://w3c.github.io/wot/current-practices/wot-practices.html`, last accessed: 08.02.2017.

[15] We use `stn:` as a prefix bound to the URI: `http://w3id.org/stn/core#`.

bring *humans in the loop* and to tackle the HATEOAS constraint. By fully conforming to REST, the STN-based abstraction layer inherits architectural properties such as *Internet-scalability* and *openness*. Therefore, the proposed abstraction layer addresses in an integrated manner all three research challenges in the engineering of agents environments that motivate our work.

4 Implementation and Evaluation

To demonstrate our approach, we implemented an STN-based agent environment for the IoT that is deployed over multiple *heterogeneous* Web services and constrained devices. Software agents running in *independent* application environments are able to seamlessly navigate and act on the underlying distributed STN in order to discover and interact with one another and with human agents.

This section is structured as follows. First, we present the application scenario in Section 4.1. Then, in Section 4.2 we present the implemented system. Finally, in Section 4.3 we discuss the workings and limitations of this proof of concept.

4.1 Application Scenario

David owns multiple smart things, such as a wristband, a mattress cover, light bulbs and curtains.[16] These things are able to produce, share and consume contextual information about David and his bedroom. For instance, when David falls asleep, both the wristband and the mattress cover can produce this information with various certainty levels and share it with the other things. David also uses a digital calendar service to keep track of important events. If there is an upcoming event scheduled and the calendar *knows* that David is still asleep, it cooperates with other things discovered at runtime in attempts to wake him up (e.g., via vibration alarms on his wristband, opening the curtains to allow natural light to enter the room). David's things are also context-aware: it makes little sense to open the curtains if the outside light level is below 100 lux (S.I.), which is the equivalent of an overcast day[17]. If all attempts fail and the scheduled event has high priority (e.g., a morning flight), the calendar crawls David's online distributed social graph in order to discover and contact friends that can wake him up. To avoid unnecessary contact attempts, the calendar must first discover which of David's friends are already awake.

In this scenario, human and software agents work together towards a common goal: to wake up David. The calendar is able to discover and interact with other agents in an *open system*. In doing so, the calendar has to navigate and use an agent environment deployed over multiple *heterogeneous platforms* (e.g., Facebook, Twitter) and constrained devices (see Figure 2), a requirement that is essential in *Internet-scale systems*. If all sotware agents' attempts to wake

[16] All connected objects used in our application scenario resemble products already available to end-users.

[17] `http://www.engineeringtoolbox.com/light-level-rooms-d_708.html`, last accessed: 08.02.2017.

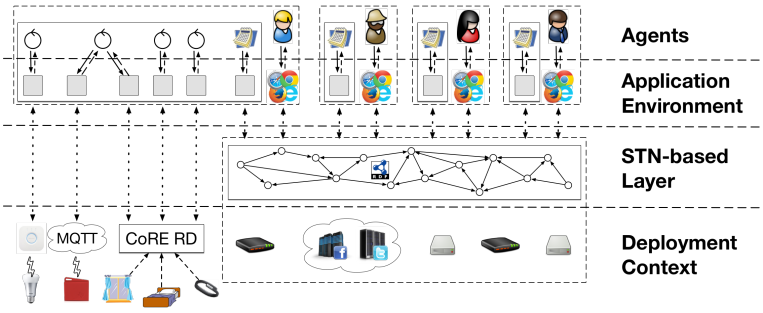

Fig. 2. Experiment setup: an STN-based agent environment deployed over multiple *heterogeneous* platforms and constrained devices.

up David fail, the calendar delegates this goal to human agents, who are thus *brought into the loop* at runtime and become part of the system. Therefore, this application scenario emphasizes all three research topics that motivate our work.

4.2 System Overview

To evaluate the above scenario, we implemented the system in Figure 2. Multiple *application environments* developed using the *JaCaMo platform* [7] are "glued" together via an STN distributed across Facebook, Twitter, and multiple instances of our own implementation of an *STN platform*. Software agents running in these application environments use the STN platforms to communicate with one another, and use Twitter to communicate with humans. We modeled each of David's things and the calendars of each of David's friends as BDI agents developed in *Jason* [8].

In what follows, we first present in further detail the *deployment context*, then the *STN-based layer*, and finally the *application environments* (see Figure 2).

Deployment Context The deployment context is *distributed* across multiple *heterogeneous* platforms and constrained devices.

We used Facebook and Twitter to deploy David's online social graphs. These platforms expose non-hypermedia APIs. The clients used in our implementation are preconfigured with the authorization tokens required to access these APIs.

To host STNs for David, each of his friends, and their things, we used multiple instances of an *STN platform*. These platforms are used to store and manage relations and as message brokers. The STN platforms expose hypermedia-driven APIs (see [4] for more technical details).

We implemented David's lights using a Philips Hue lightbulb. The lightbulb communicates with a bridge via ZigBee Light Link, and the bridge exposes a non-hypermedia HTTP API for controlling the lightbulb.[18]

[18] For more details: `http://www.developers.meethue.com/documentation/how-hue-works`, last accessed: 08.02.2017.

We implemented the outside light sensor using a Texas Instruments (TI) SensorTag. The TI SensorTag communicates with a smartphone via Bluetooth Low Energy, which in turn pushes the sensor readings to a cloud-based MQTT broker[19]. MQTT clients can then subscribe to this broker to receive real time sensor readings (see details in Section 4.2).

To implement the wristband, mattress cover, and window curtains, we emulated CoAP devices using the Californium (Cf) framework [26]. Following the typical setup used in *constrained RESTful environments (CoRE)*, these devices register their resources with a CoRE Resource Directory[20], which allows them to be discovered dynamically at runtime. We used Cf-RD[21] to deploy the resource directory.

STN-based Layer In our experiment setup, David has an `stn:connectedTo` social relation to a friend on his STN platform, two other friends on Facebook, and follows two users on Twitter. We used Facebook's test harness and regular Twitter accounts to set up David's social graphs on these platforms.

The description of David's user account on his STN platform advertises the user accounts he `stn:holds` on Facebook and Twitter, which makes them discoverable and allows agents to *navigate into* these networks. To enable agents to *navigate out* of Facebook and Twitter, David's friends advertise their URIs as their *personal websites*, an attribute field available on most social platforms.

Once all data is lifted to RDF (see [3] for technical details), software agents have a uniform RDF-based view of the distributed STN and are *decoupled* from its underlying *heterogeneous platforms* (see Figure 2).

Application Environments The application environments programmed for this scenario provide agents with *CArtAgO artifacts* [34] that they can use to manipulate STNs and to interact with devices.

The *STN artifacts* are environment abstractions that encapsulate parsers for interpreting RDF descriptions of STN platforms, and HTTP clients used to issue requests constructed *at runtime* based on these descriptions. The platform descriptions are *discovered at runtime* via the `stn:hostedBy` relation and provide all the knowledge required to interface with the platforms, which includes metadata about the platform's API (e.g., supported authentication protocols and media types) and descriptions of supported STN operations.

The *device artifacts* provide agents with wrappers for devices in their deployment context. The Philips Hue artifact is a wrapper over an HTTP client that accesses the Philips Hue bridge in the local network, whereas the TI SensorTag artifact is a wrapper over an MQTT client that communicates with an MQTT broker in the cloud. Similarly, the artifacts for CoAP devices are wrappers over CoAP clients that access the emulated devices. These artifacts are

[19] In our implementation, we used HiveMQ: `http://www.hivemq.com/`, last accessed: 08.02.2017.

[20] `https://tools.ietf.org/html/draft-ietf-core-resource-directory-09`

[21] `https://github.com/eclipse/californium.tools`, last accessed: 08.02.2017.

created dynamically at runtime based on the *resource types* discovered via the CoRE resource directory.

4.3 Proof of Concept

The application scenario runs in multiple stages. First, David's agents get bootstrapped into the STN layer via preprogrammed behavior. Their *entry point* is David's URI, which they use to declare David as their owner by creating `stn:ownedBy` relations. In doing so, the agents are acting on the distributed STN in an *autonomous* and *reliable* manner in order to *make themselves discoverable*. Then, they can crawl the STN to discover and subscribe to to all other agents `stn:ownedBy` David.

Once interconnected, agents interact using the FIPA Contract Net Protocol [17]: the *calendar* agent publishes a call for proposals to wake up David via his STN platform, and the *wristband*, *lights* and *window curtains* agents reply with proposals. The agents use a shared communication language (in our implementation, the one provided by Jason) and a shared vocabulary for describing the state of the environment. The window curtains agent decides to join the interaction based on the readings of the outside light sensor. All wake-up attempts eventually fail in our implementation.

Next, the calendar agent crawls David's online distributed social graph to discover friends that also `stn:owns` calendar agents, where the *entry point* in the distributed STN is once again David's URI. We described this crawling process in detail for a similar scenario in [3]. Once David's calendar discovers the other calendar agents in our system (see Figure 2), it asks each of them if their owner is asleep. For each friend that is awake, the calendar then searches for an `stn:UserAccount` they `stn:hold` on an `stn:Platform` that implements the `stn-ops:SendDirectMessage` operation. The only friend who satisfies this criteria is Mike, who is awake and holds an account on Twitter, which implements the required operation. The calendar sends Mike a direct message on Twitter to wake up David.

This proof of concept application demonstrates that we can successfully apply our approach to engineer agent environments that address the three research topics that motivate our work in an integrated manner. Note that the agent environment used in this application integrates two of the most used online social platforms in a seamless manner.

The *STN artifacts* are decoupled from the platforms that host the distributed STN in our experiment. The *device artifacts* are still tightly coupled to the deployment context (see Figure 2), but they are hidden behind agents. However, these artifacts can also be decoupled using the same approach. First, they have to be linked in the distributed STN to become discoverable, and then they have to translate the higher-level semantics of operations defined by a domain-specific model (e.g., a general model for light bulbs) to the lower-level semantics of a WoT protocol used by the devices' APIs.

5 Conclusions

The World Wide Web is arguably the most scalable and versatile software system deployed on the Internet [13], and most other software systems today revolve around the Web. This huge success comes from the architectural properties of Web, which was specifically designed as an Internet-scale and long-lived system in which components can be deployed and can evolve independently from one another. Precisely because of its design goals, key architectural principles and ubiquity in people's lives, in this paper we claimed that the Web provides a suitable middleware for agent environments that (i) support large scale systems, (ii) can cope with open systems, and (iii) support humans in the loop, three important and current research topics in the engineering of agent environments [38].

Our approach is to apply the REST architectural style to design a resource-oriented abstraction layer for agent environments that decouples the *application environment* from its *deployment context*. Higher-level environment abstractions can then be implemented on top of this lower-level abstraction layer. The novelty of our approach is the use of *socio-technical networks (STNs)* as a means to address the HATEOAS constraint, which is one of the core tenets of the modern Web architecture. By analogy with how the Web enables the discovery of Web pages, STNs enable the discovery of agents in agent environments on the Web.

We consider the impact of the contribution presented in this paper to be twofold. On the one hand, it addresses important research topics in engineering MAS. On the other hand, it enables the transfer of MAS technology to the development of Web-based systems in a manner that is completely aligned with the Web architecture. We consider the latter to be an important step towards achieving the vision of a *Semantic Web*, as originally described in [6].

References

1. Abdullah Althagafi. Designing a Framework for RESTful Multi-Agent Systems. Master's thesis, Department of Computer Science, University of Saskatchewan, Saskatoon, Canada, 2012.
2. Andrei Ciortea, Antoine Zimmerman, Olivier Boissier, Adina Magda Florea. Towards a Social and Ubiquitous Web: A Model for Socio-technical Networks. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 1, pages 461–468. IEEE, 2015.
3. Andrei Ciortea, Antoine Zimmerman, Olivier Boissier, Adina Magda Florea. Hypermedia-driven Socio-technical Networks for Goal-driven Discovery in the Web of Things. In *Proceedings of the 7th International Workshop on the Web of Things (WoT)*. ACM, 2016.
4. Andrei Ciortea, Olivier Boissier, Antoine Zimmerman, Adina Magda Florea. Responsive Decentralized Composition of Service Mashups for the Internet of Things. In *Proceedings of the 6th International Conference on the Internet of Things (IoT)*. ACM, 2016.
5. Estefania Argente, Vicente Botti, Carlos Carrascosa, Adriana Giret, Vicente Julian, and Miguel Rebollo. An abstract architecture for virtual organizations: The thomas approach. *Knowledge and Information Systems*, 29(2):379–403, 2011.

6. Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):28–37, 2001.
7. Olivier Boissier, Rafael H Bordini, Jomi F Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Science of Computer Programming*, 78(6):747–761, 2013.
8. Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons, 2007.
9. Lars Braubach and Alexander Pokahr. Conceptual Integration of Agents with WSDL and RESTful Web Services. In *International Workshop on Programming Multi-Agent Systems*, pages 17–34. Springer, 2012.
10. Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. W3C Recommendation, World Wide Web Consortium (W3C), February 25 2014.
11. Oguz Dikenelli. SEAGENT MAS platform development environment. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers*, pages 1671–1672. International Foundation for Autonomous Agents and Multiagent Systems, 2008.
12. Jose Exposito, Joan Ametller, and Sergi Robles. Configuring the JADE HTTP MTP. http://jade.tilab.com/documentation/tutorials-guides/configuring-the-jade-http-mtp/, 2010. Accessed: 15.11.2016.
13. Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.
14. Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
15. Foundation for Intelligent Physical Agents. FIPA Abstract Architecture Specification. http://www.fipa.org/specs/fipa00001/SC00001L.html, 2002. Document number: SC00001L.
16. Foundation for Intelligent Physical Agents. FIPA Agent Message Transport Protocol for HTTP Specification. http://www.fipa.org/specs/fipa00084/SC00084F.html, 2002. Document number: SC00084F.
17. Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. http://www.fipa.org/specs/fipa00029/SC00029H.html, 2002. Document number: SC00029H.
18. Nicholas Gibbins, Stephen Harris, and Nigel Shadbolt. Agent-based semantic web services. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(2):141–154, 2004.
19. Abdelkader Gouaïch and Michael Bergeret. Rest-a: An agent virtual machine based on rest framework. In *Advances in Practical Applications of Agents and Multiagent Systems*, pages 103–112. Springer, 2010.
20. Dominic Greenwood and Monique Calisti. Engineering web service-agent integration. In *Systems, Man and Cybernetics, 2004 IEEE International Conference on*, volume 2, pages 1918–1925. IEEE, 2004.
21. Dominic Greenwood, Margaret Lyell, Ashok Mallya, and Hiroki Suguri. The ieee fipa approach to integrating software agents and web services. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 276. ACM, 2007.
22. Miguel Escrivá Gregori, Javier Palanca Cámara, and Gustavo Aranda Bada. A jabber-based multi-agent system platform. In *Proceedings of the fifth international*

joint conference on Autonomous agents and multiagent systems, pages 1282–1284. ACM, 2006.
23. Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010.
24. Michael N Huhns. Agents as web services. *IEEE Internet computing*, 6(4):93, 2002.
25. Michael N Huhns and Munindar P Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet computing*, 9(1):75–81, 2005.
26. Matthias Kovatsch, Martin Lanter, and Zach Shelby. Californium: Scalable cloud services for the internet of things with coap. In *Internet of Things (IOT), 2014 International Conference on the*, pages 1–6. IEEE, 2014.
27. Markus Lanthaler and Christian Gütl. Hydra: A vocabulary for hypermedia-driven web apis. *Proceedings of the 6th Workshop on Linked Data on the Web*, 996, 2013.
28. Dejan Mitrović, Mirjana Ivanović, Zoran Budimac, and Milan Vidaković. Radigost: Interoperable web-based multi-agent platform. *Journal of Systems and Software*, 90:167–178, 2014.
29. Xuan Thang Nguyen and Ryszard Kowalczyk. Ws2jade: Integrating web service with jade agents. In *International Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering*, pages 147–159. Springer, 2007.
30. Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous agents and multi-agent systems*, 17(3):432–456, 2008.
31. Benno J Overeinder and Frances MT Brazier. Scalable middleware environment for agent-based internet applications. In *International Workshop on Applied Parallel Computing*, pages 675–679. Springer, 2004.
32. Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big" web services: Making the right architectural decision. In *Proceedings of the 17th international conference on World Wide Web*, pages 805–814. ACM, 2008.
33. Alessandro Ricci, Enrico Denti, and Michele Piunti. A platform for developing soa/ws applications as open and heterogeneous multi-agent systems. *Multiagent and Grid Systems*, 6(2):105–132, 2010.
34. Alessandro Ricci, Mirko Viroli, and Andrea Omicini. CArtAgO: A framework for prototyping artifact-based environments in MAS. In *Environments for Multi-Agent Systems III*, pages 67–86. Springer, 2007.
35. Michael Rovatsos, Dimitrios Diochnos, and Matei Craciun. Agent protocols for social computation. In *Advances in Social Computing and Multiagent Systems*, pages 94–111. Springer, 2015.
36. M Omair Shafiq, Ying Ding, and Dieter Fensel. Bridging multi agent systems and web services: towards interoperability between software agents and semantic web services. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, pages 85–96. IEEE, 2006.
37. Munindar P Singh and Michael N Huhns. *Service-oriented computing: semantics, processes, agents*. John Wiley & Sons, 2006.
38. Danny Weyns, Fabien Michel, et al. Agent environments for multi-agent systems – a research roadmap. In *Lecture Notes in Computer Science*, volume 9068, pages 3–21. Springer, 2015.
39. Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous agents and multi-agent systems*, 14(1):5–30, 2007.
40. Erik Wilde. Putting things to rest. *School of Information, UC Berkeley*, 2007.