# Responsive Decentralized Composition of Service Mashups for the Internet of Things

**Andrei Ciortea**
Univ. Lyon, MINES Saint-Étienne, CNRS
Lab Hubert Curien UMR 5516
F-42023 Saint-Étienne, France
andrei.ciortea@emse.fr

**Olivier Boissier**
Univ. Lyon, MINES Saint-Étienne, CNRS
Lab Hubert Curien UMR 5516
F-42023 Saint-Étienne, France
olivier.boissier@emse.fr

**Antoine Zimmermann**
Univ. Lyon, MINES Saint-Étienne, CNRS
Lab Hubert Curien UMR 5516
F-42023 Saint-Étienne, France
antoine.zimmermann@emse.fr

**Adina Magda Florea**
University Politehnica of Bucharest
Bucharest, Romania
adina.florea@cs.pub.ro

## ABSTRACT

Applications envisioned for the Internet of Things (IoT) would generally have to fulfill their design goals by mashing up devices and digital services in a manner that is both *flexible*, such that they can adapt to dynamic environments, and *responsive*, such that they can react to sensor and user input in a timely fashion. Most existing approaches for the development of IoT applications rely on precompiled mashups that are highly responsive, but inflexible due to their static nature. At the other end of the spectrum, fully automatic composition of services results in IoT mashups that are highly flexible, but responsive only for small numbers of IoT services. This paper presents a middle ground approach: goal-driven software agents are equipped with precompiled mashups and cooperate with one another to compose their mashups at runtime in pursuit of their goals. Agents are interconnected via relations that enable them to discover and interact with one another in a flexible manner. To support our approach, we provide an open-source platform that facilitates application development. We used this platform to implement a realistic IoT application that achieves its design goal by mashing-up multiple heterogeneous devices at runtime. Evaluation results suggest that applications remain responsive when scaling to many devices and for relatively large mashup compositions.

## ACM Classification Keywords

H.3.5. Online Information Services: Web-based services; I.2.11. Distributed Artificial Intelligence: Multiagent systems



*IoT'16,* November 07-09, 2016, Stuttgart, Germany

 ISBN 978-1-4503-4814-0/16/11…$15.00
DOI: http://dx.doi.org/10.1145/2991561.2991573

## Author Keywords

Web of Things; intelligent agents; service mashups.

## INTRODUCTION

Applications for the Internet of Things (IoT) would generally have to fulfill predefined or user-specified goals by composing services provided by resource-constrained devices with digital services. The challenge, however, is that IoT environments are expected to be populated by *large numbers* of *heterogeneous* devices that may become available or unavailable *at runtime*. To accommodate this vision, IoT applications would have to be both *flexible*, to adapt to dynamic networks of heterogeneous devices, and *responsive*, to react to user input and relevant events in a timely manner even when scaling to many devices.

However, existing approaches to IoT application development are usually polarized either towards responsiveness or flexibility. Most approaches rely on precompiled service mashups created manually by developers and end-users, which results in IoT applications that are highly responsive, but inflexible due to their static nature. At the other end of the spectrum, fully automatic composition of services at runtime results in IoT applications that are highly flexible, but the service composition overhead grows exponentially with the number of IoT services in the environment.

To emphasize both application flexibility and responsiveness, we developed a middle ground approach that relies on the decentralized goal-driven composition of precompiled service mashups: goal-driven software agents are equipped with libraries of *plans* that they use to achieve their goals, and cooperate with one another to compose IoT mashups at runtime in pursuit of their goals. Agents are interconnected in dynamic network-like structures that enable them to autonomously discover and interact with one another. Evaluation results show that, in the worst-case scenario, the mashup composition overhead grows only linearly with the number of agents, which would typically be less than or equal to the number of devices in the environment.

### Application Scenario: The Wake-up Call

To illustrate and motivate our approach, we introduce a concrete application scenario that we use throughout the rest of this paper. In this scenario, David owns multiple connected objects, such as a wristband, a mattress cover, light bulbs and curtains.[1] These objects are able to produce, share and consume contextual information about David and his bedroom. For instance, if David is sleeping, both the wristband and the mattress cover can produce this information with various confidence levels and share it with David's other things. In his daily life, David uses an online calendar service to keep track of important events. The calendar service, however, can also interact with David's connected objects, for instance, to wake up David whenever he is still sleeping and there is an upcoming event scheduled, such as a morning flight. Furthermore, the calendar service is able to *decide* on various ways to wake up David: it usually starts with smoother attempts, such as vibration alarms via David's wristband, and escalates to more intrusive attempts, such as opening the curtains to allow natural light to enter the room or sound alarms via David's smartphone. The calendar's decisions are also contextual: there is little reason to open the curtains or turn on the lights in David's bedroom if it is *not known* that David is sleeping at home (e.g., the wristband signals that David is sleeping, but the mattress cover does not). If a wake-up attempt is successful, David's wristband and mattress cover can both signal the event.

In this scenario, David's things are goal-driven agents that interact with one another in pursuit of a common goal, that is to wake up David. The calendar agent leads the interaction: it makes decisions based on contextual information produced and shared by the other agents, it delegates the goal of waking up David and monitors its achievement. The service mashup is composed with the cooperation of agents available at runtime and executed in a decentralized manner.

### Contribution and Paper Outline

Our main contribution is a novel approach to IoT mashup composition that emphasizes both the flexibility and responsiveness of resulting applications. Sec. 2 introduces the foundation on which we built our proposal and further motivates our work. Sec. 3 describes in detail our approach to IoT mashup composition. Sec. 4 presents an implementation of this approach. Sec. 5 provides a quantitative evaluation of the responsiveness of resulting IoT applications, and an evaluation of application flexibility via implementing "The Wake-up Call" scenario.

## BACKGROUND AND RELATED WORK

On account of its scalable and flexible architecture, the World Wide Web is emerging as the application layer for the IoT, or the so-called Web of Things (WoT) [27, 10]. The underlying idea is to apply the REST architectural style to design IoT environments that integrate seamlessly into the Web. Devices and digital services converge at the application layer via RESTful Web APIs that hide component heterogeneity behind uniform interfaces designed using URIs, Web transfer protocols and standard media types. Standardization efforts led by the Internet Engineering Task Force (IETF) are rapidly turning this vision into reality [11]. Most notably, the Constrained Application Protocol (CoAP) [25] enables the direct integration into the Web of devices with as little as 100 KiB of ROM and 10 KiB of RAM [13].

The WoT enables developers to mash-up devices with digital services using familiar Web technologies [9, 15]. To further ease IoT application development, several mashups editors have been proposed both in the academia [1, 12, 8] and the industry[2]. These tools are domain-independent and provide developers with visual abstractions of devices and services that they can wire together. Other mashup tools target end-users and specific application domains, such as home automation. With homeBlox [22], for instance, end-users create mashups by connecting graphical abstractions of human activities, household appliances, logical operators etc.

The above tools follow the dataflow programming paradigm. Another category of tools gaining traction recently are cloud-based services that enable end-users to create IoT mashups via event-condition-action rules, such as IFTTT[3] or Zapier[4]: end-users define rules that are activated by a triggering event and execute one or more actions. Rules can be chained to create more complex mashups.

These approaches to IoT application development result in applications that are highly responsive: IoT mashups are created manually by developers and end-users and thus there is no mashup composition overhead at runtime. The tradeoff, however, is that resulting applications rely on static mashups that cannot adapt to services becoming available or unavailable at runtime. Furthermore, manually wiring the IoT is cumbersome to everyday users and impractical when dealing with many heterogeneous devices.[5]

In a different approach, hypermedia-driven interaction is used to achieve a more flexible execution of IoT mashups [18]: developers publish hyperlinks between related services to create graphs that *mashup clients* can navigate using various pre-programmed strategies. The hyperlinks are annotated with meta-information (e.g., forward path name, service cost) to provide local guidance to clients in their traversal. To enable global guidance, hyperlinks can also be annotated with the name of the mashup that the client is currently traversing. Mashups are still created manually, but flexibility is achieved via path selection. This approach may be a good candidate for developing IoT applications that are both flexible and responsive, but an evaluation of application responsiveness is not available.

The automatic composition of services, a highly researched topic [26], has also been explored as a means to compose IoT mashups at runtime, for instance in [17, 14]: a central reasoner

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

[2] http://www.nodered.org/, Accessed: 27.06.2016.
[3] http://www.ifttt.com/, Accessed: 27.06.2016.
[4] http://www.zapier.com/, Accessed: 27.06.2016.
[5] http://www.ericsson.com/uxblog/2012/04/a-social-web-of-things/, Accessed: 27.06.2016.

composes functional semantic descriptions of services to create execution plans that achieve the mashup's design goal. Automated planning, however, is computationally costly [7]. Resulting IoT applications are highly flexible, but the mashup composition overhead grows exponentially with the number of stateful IoT services involved in the process [14].

We conclude that most existing approaches to IoT application development are polarized either towards application responsiveness or flexibility, while approaches that emphasize both characteristics are insufficiently investigated.

## A DECENTRALIZED APPROACH TO SERVICE MASHUP COMPOSITION FOR THE IOT

In this section, we present a decentralized approach to IoT mashup composition. This approach relies on goal-driven software agents equipped with libraries of precompiled plans that they use to achieve their goals. If an agent cannot achieve its goals by itself, it cooperates with other agents at runtime. To be able to discover one another, agents are interconnected via relations that they can crawl.

In the following, we first define these network-like systems, which we call *socio-technical networks (STNs)*, that enable discoverability and provide the underpinning of our approach. We then further detail how agents are able to cooperate in a loosely coupled manner in order to compose IoT mashups at runtime.

### Socio-technical Networks

A *socio-technical network (STN)* is a dynamic system of people and things interrelated in a meaningful manner via typed relations (e.g., friendship, ownership, provenance, colocation). People and things can enter or leave the STN, manipulate their relations to rewire the network, or interact with one another via messages.

We use STNs as a means to model IoT environments and then build applications on top of these abstractions. To facilitate application development, STNs can be hosted on Web platforms concerned with storing relations and routing messages based on those relations. To support a uniform interface between STN applications and platforms, we provide a Web ontology[6], called hereafter the *STN ontology*, that developers can use to describe STNs in the *Resource Description Framework (RDF)*. Having a uniform interface enables STNs to be seamlessly distributed across multiple Web platforms.

In what follows, we first discuss modeling IoT environments as STNs, and then using the STN ontology to describe IoT environments in a uniform manner.

#### *Modeling IoT environments*

STNs model IoT environments via three primary abstractions: *agents*, *artifacts* and *relations* among them (see Fig. 2).[7]

People and things that are actively trying to influence the state of the environment are modeled as *agents*. Things that passively augment the environment with new capabilities are modeled as *artifacts*. For instance, David's online calendar is proactive in waking up David and can delegate goals to other things, and the lights and curtains can manipulate the state of David's bedroom. David, his calendar, lights and curtains can be modeled as agents. In contrast, David's mattress cover is more passive, it can only signal when David is going to bed or waking up, and can be modeled as an artifact that agents can observe. Note, however, that abstracting a thing as either an agent or an artifact is a choice made at the modeling level, and not an intrinsic property of the thing itself.

The agent and artifact abstractions are motivated by the separation of concerns principle and bring several benefits. First, these abstractions separate exhibited behavior from the actual entities, which enables developers and end-users to conceive of people and heterogeneous things in a uniform manner. Second, agents and artifacts simplify the design of IoT applications by separating the logic that manipulates the environment from the logic that augments the environment. Third, agents and artifacts provide a modular approach to IoT application development when designed as loosely coupled components meant to be deployed and to evolve independently at runtime.

As introduced previously, agents and artifacts are interconnected in STNs via *typed relations*, which enable discoverability and support reasoning. For instance, David owns multiple things, which is represented in his home STN via *ownership relations*. Software agents can crawl the STN in an informed manner to discover all things *owned by* David.

#### *Describing IoT environments*

*Agents*, *artifacts* and *relations* among them are represented on STN platforms by means of *digital artifacts*, that is Web resources described using the STN ontology. For instance, we assume David's home STN is hosted on an STN platform running on his WoT hub. David and his agents are represented on the STN platform via *user accounts*, which are digital artifacts that they can use as proxies in the STN to establish relations and exchange messages with one another.

The STN ontology describes a general model for STNs[8] that defines common types of *digital artifacts* (e.g., user accounts, digital messages, digital groups) and *relations*, such as:[9]

- `stn:connectedTo`, which denotes a general unidirectional relation between two agents (or their user accounts);
- `stn:ownedBy`, which denotes a unidirectional ownership relation from a thing to its owner (e.g., a person, a group);
- `stn:subscribedTo`, which denotes a unidirectional relation between two agents (or their user accounts) that is used for communication.

Based on these relations (and possible extensions), developers can program agents to autonomously discover and interact with one another. For instance, when registering to David's STN platform, his agents declare David as their owner via

[6] https://w3id.org/stn/core

[7] A formal mathematical definition of STNs is available in [4].

[8] Developers can further extend the STN ontology with domain- and application-specific knowledge.

[9] We use the `stn:` prefix throughout the rest of this paper to denote the namespace: `http://w3id.org/stn/core#`.

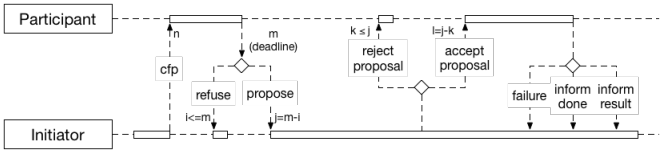


**Figure 1. The FIPA Contract Net Interaction Protocol (sourced from [6]): an initiator launches a call for proposals for a task to be achieved, awaits for proposals for a predefined amount of time, and then informs participants of the result of the interaction.**

an `stn:ownedBy` relation, which makes them discoverable. Agents can then crawl David's socio-technical graph to create `stn:subscribedTo` relations to all other agents he owns. Once relations are established, agents can use the STN platform as a central broker that routes messages based on their relations.

### IoT Mashup Composition

In this section, we first present the IoT mashup composition mechanism that relies on agents cooperating with one another at runtime, and then discuss how agents interact in a loosely coupled manner. Agents discover and interact with one another via STN platforms, such as the one described for David's home.

#### *Goal-driven mashup composition*

Agents are goal-driven and achieve their goals by executing precompiled plans. In doing so, agents use various devices and digital services, but they may not hold all the resources they need to achieve their goals. To address this problem, agents cooperate with one another and compose IoT mashups at runtime in pursuit of their goals.

The composition mechanism relies on *goal decomposition trees*, in which a goal is decomposed into sub-goals and can only be achieved after all of its sub-goals have been achieved. A goal decomposition tree is distributed across all the agents that participate to its achievement and it is composed dynamically using preprogrammed goal decompositions.

For illustrative purposes, in our application scenario, the design goal of the calendar agent is *to wake up David if there is an upcoming event and David is asleep*. This goal can be further decomposed in three sub-goals:

- *to determine if there is an upcoming event*;
- *to determine if David is asleep*;
- if the case, *to wake up David*.

The calendar agent can use an online calendar service to achieve the first goal, but it cannot use sensors or actuators to achieve the other goals. However, the calendar agent can ask the other agents if David is asleep or who can wake him up via the STN platform.

In the above example, sub-goals are achieved on-demand, but they can also be achieved proactively. For instance, instead of waiting for other agents to ask if David is asleep, the wristband agent can proactively inform all of its subscribers on the STN platform whenever David falls asleep or wakes up.

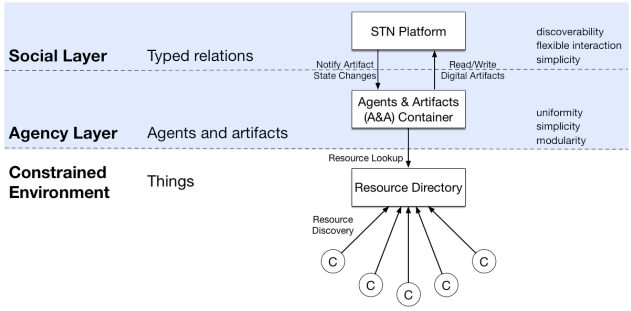


**Figure 2. This image illustrates the main system components of our implementation and the core abstractions introduced to model IoT environments as STNs. Devices discovered at runtime are abstracted as *agents* and *artifacts* interconnected via *typed relations* that they can crawl and manipulate. Agents use STN platforms to autonomously discover and interact with one another in pursuit of their goals. The layers shown here are part of a layered architecture we have defined for a *social WoT* [4].**

#### *Agent interactions*

In order to cooperate, agents must be able to interact in an autonomous and reliable manner. Furthermore, in an open system in which no prior assumptions can be made about the design and implementation of agents, interactions have to rely on standardized knowledge.

Noticeable efforts have been undertaken to standardize communication and interaction in multi-agent systems[10] and various collections of standards are publicly available to specify, among others:[11]

- *agent communication languages* that define the structure and semantics of messages exchanged between agents;
- various *interaction protocols* that define sequences of messages exchanged between agents.

For instance, the calendar agent can implement the FIPA Contract Net Interaction Protocol [6], depicted in Fig. 1, to delegate the goal of waking up David. It would first launch a *call for proposals* via the STN platform in order to discover who is available to achieve this goal. Some of David's agents may reply with proposals of various alarms they can trigger (e.g., vibration alarms, sound alarms). The calendar agent then decides what proposals to *accept* or *reject* (if any) based on David's preferences. We discuss our implementation of this interaction in Sec. 5.2.

## SOCIO-TECHNICAL NETWORKS FOR CONSTRAINED RESTFUL ENVIRONMENTS

We present an implementation of STNs for constrained RESTful environments (CoRE) that relies on a *CoRE Resource Directory* [23], a multi-agent platform, and our own implementation of an STN platform.

Fig. 2 depicts an overview of the main system components. CoAP devices register with the resource directory, and an *Agents & Artifacts (A&A) container* synchronizes with the

[10] Such as the ones undertaken by the Foundation for Intelligent Physical Agents (FIPA): http://www.fipa.org/, Accessed: 27.06.2016.

[11] http://www.fipa.org/repository/standardspecs.html, Accessed: 27.06.2016.

resource directory to create or delete agents and artifacts as devices become available or unavailable at runtime. Agents use an *STN platform* to create and participate in STNs.

### Constrained RESTful Environment

We use the Californium (Cf) framework [16] to emulate multiple CoAP devices and Cf-RD[12] to deploy the resource directory. Devices register with the resource directory via a standard CoAP interface [23]: each device sends a `POST` request to a predefined endpoint with a list of provided resources in the CoRE Link Format [24].

Devices can register one or more application-specific resource types for each of their provided resources. In our implementation, we rely on resource types and shared ontologies to decouple the A&A container from devices (see Sec. 4.3 for details).

### STN Platform

The STN platform provides[13] two main functionalities:

- it acts as a repository that agents can use to store and query representations of digital artifacts, and
- it dispatches notifications to agents when states of observed digital artifacts change.

The platform implements an event-driven non-blocking architecture and is built on top of *Vert.x*[14], a polyglot framework for the Java Virtual Machine that is both powerful enough to support high-throughput Web servers[15], and lightweight enough to perform well on small devices, such as Raspberry Pi[16].

The platform exposes a RESTful HTTP interface for reading and writing representations of digital artifacts. Digital artifacts are uniformly identified via URIs and their state is represented in RDF using the STN ontology. The RESTful interface decouples software clients from the STN platform, which facilitates the deployment of STNs distributed across multiple instances of the STN platform.

For illustrative purposes, an agent registers to the platform by sending a `POST` request to the `/users/` endpoint that encloses in its body a Turtle [20] representation of the *user account* to be created as shown in Listing 1. If the operation is successful, the STN platform responds with a `201 Created` status code and a `Location` header field with the URI of the created user account. Registration is therefore completely dynamic, and the agent can store and use the URI of its user account for subsequent requests.

Agents can register callback HTTP URIs to receive notifications from the STN platform, for instance, when other agents create relations to them or when they receive messages. Messages are routed based on relations in the STN. If an agent creates a message that has one or more explicit receivers, the STN platform dispatches notifications to the specified receivers. Otherwise, the message is broadcast to all agents that have a relation of type `stn:subscribedTo` to the sender of the message. Notifications are sent by `POST`ing the current states of observed artifacts to callback URIs.

**Listing 1. A sample HTTP request/response for creating a user account on the STN platform. The null relative URI in the request payload is used to identify the artifact to be created. The response returns the URI of the created user account via the `Location` header field.**

```
POST /users/ HTTP/1.1
Host: localhost:8080
Content-Type: text/turtle

@prefix stn: <http://w3id.org/stn/core#> .

<> a stn:UserAccount ;
     stn:callbackUri <http://localhost:58880/calendar/> ;
     stn:heldBy <http://example.org#calendar> .

<http://example.org#calendar> a stn:SocialThing ;
   stn:ownedBy <http://localhost:8080/users/fae5(...)5a26> .


HTTP/1.1 201 Created
Location: http://localhost:8080/users/b770(...)b21b
Content-Length: 0
```

### Agents & Artifacts Container

The *Agents & Artifacts (A&A) container*[17] is a platform for programming and running agents and artifacts in the STN. Our implementation uses JaCaMo [2], a multi-agent platform for the development of *Belief-Desire-Intention (BDI) agents* [3] (i.e., agents that can decide and act on their own) and artifact-based multi-agent environments [21].

Agents are developed using *Jason* [3], a framework that provides a customizable BDI agent architecture and a language for programming agent behavior in terms of *beliefs* held about the world, *goals* desired to be achieved, and *plans* used to achieve goals. An important feature of *Jason agents* that make them a good fit for our approach is that they are both *goal-driven* and *reactive*: agents commit to goals by executing plans, but they can still react to new stimuli from the environment while executing their plans.

The A&A container uses the resource directory to discover resources of known types in the environment, and then instantiates agents and artifacts based on those types. The instantiation logic is programmed by the IoT application developer. In our application scenario, for instance, if the A&A container discovers a resource of type `ex:SmartWristband`[18], it instantiates an agent that implements preprogramed behavior associated to a smart wristband. The A&A container ignores any resources of unspecified or unknown types.

We assume the A&A container and the devices rely on shared ontologies to interpret resource types, and to exchange and interpret resource representations in a reliable manner.

[12] http://github.com/eclipse/californium.tools, Accessed: 27.06.2016.

[13] https://github.com/andreiciortea/stn-platform

[14] http://www.vertx.io/, Accessed: 27.06.2016.

[15] According to independent benchmarks for Web frameworks: https://www.techempower.com/benchmarks/#secOon=data-r8&hw=i7&test=plaintext, Accessed: 27.06.2016.

[16] http://vertx.io/blog/vert-x3-web-easy-as-pi/, Accessed: 27.06.2016.

[17] https://github.com/andreiciortea/stn-agents-iot16

[18] The `ex:` prefix denotes the namespace `http://example.org#`.

The A&A container can interface with the STN platform via both HTTP and the WebSocket protocol [5]. We use the latter in our quantitative evaluation of application responsiveness in order to minimize integration latency.

## EVALUATION

In this section, we evaluate the *responsiveness* and *flexibility* of STN-based IoT applications. In Sec. 5.1 we provide a quantitative evaluation of application responsiveness, and in Sec. 5.2 we assess the flexibility of an STN-based implementation of the application scenario presented in Sec. 1.1.

### Application Responsiveness

To evaluate the responsiveness of STN-based IoT applications, we measured the mashup composition overhead introduced by agent interactions. The overhead for executing the resulting mashups is independent of how mashups are created and not considered here. We assume that all agent relations in our experiments have been established in advance and do not consider the overhead for creating relations, which we consider a one-time cost for the purpose of this evaluation and independent of the applications using the resulting STN.

#### *Evaluation method*

IoT mashups are composed via independent interactions among agents. In the worst-case scenario, an agent would have to interact with all other agents in the STN to achieve its purpose. We thus expect that, in this worst-case scenario, the mashup composition overhead grows linearly with the number of agents in the STN. A first objective of our responsiveness evaluation is to confirm this assumption.

Our second objective is to estimate the number of sequential interactions that can be performed in a typical STN in less than 1 second. We consider 1 second to be a reasonable mashup composition latency for IoT applications that would classify as responsive in most use-case scenarios. We use the work presented in [14] as a reference to estimate about 250 software agents in a typical STN.

We evaluated worst-case performance by implementing the FIPA Contract Net Interaction Protocol (see Fig. 1) in a setting in which all agents in the STN are subscribed to a central agent that launches the call for proposals. All agents respond and we measure the time spent from launching the call to receiving all proposals. We ran our experiments with 10, 50, 100 and 250 participants to evaluate performance in a typical STN, and with 500, 750 and 1000 participants to observe system behavior beyond that threshold.

#### *Experiment setup*

We ran our experiments on a laptop computer with Intel(R) Core(TM) i7-3667U CPU @ 2.0GHz (2 physical cores) in four different setups:

In the first setup (`A&A-2c`), we used only the *A&A container* running on both cores. The purpose of this setup is to evaluate the overhead introduced by running BDI agents: agents do not use STNs, the system is closed and the initiator broadcasts the call for proposals to all agents via JaCaMo's built-in communication infrastructure.

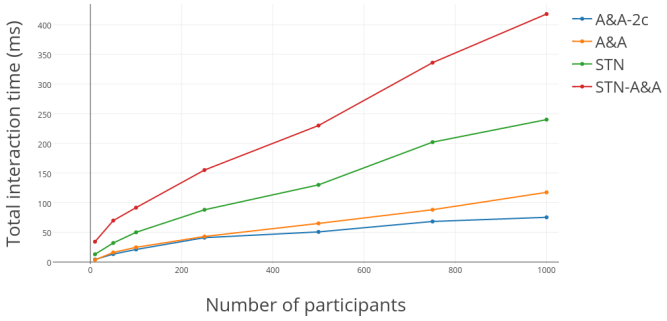


**Figure 3. These results measure the time required for a Contract Net-like interaction with an increasing number of participating agents.**

**Table 1. Number of contract-net-like interactions with 250 and 1000 participants completed within 1 second.**

| # participants | A&A-2c | A&A | STN | STN-A&A |
|---|---|---|---|---|
| **250** | 24 | 23 | 11 | 6 |
| **1000** | 13 | 8 | 4 | 2 |

The second setup (`A&A`) was identical with the previous one except we ran it on a single core.

In the third setup (`STN`), we used only the *STN platform* with mock-up agents running directly on the platform. The purpose of this setup is to evaluate the overhead introduced by the *STN platform* for routing messages in the STN.

In the fourth setup (`STN-A&A`), we used both the *STN platform* and the *A&A container* communicating via the WebSocket protocol to minimize integration latency.

The third and fourth setups use a single core due to implementation limitations at the moment of writing this paper.

#### *Results*

For each of the three setups, we repeated each interaction 10 times to obtain an average. Results are shown in Fig. 3. The plotted series are approximately linear, which confirms our assumption that, in the worst-case scenario, the mashup composition overhead grows linearly with the number of agents.

The various slopes of the plotted series indicate the implementation performance of each of the four setups. The `A&A-2c` setup, running on both physical cores, performed best. The `STN` setup performed worse than the `A&A` setup, and the greater slope seems to indicate an implementation overhead that increases linearly with the number of agents in the interaction. Intuitively, we assume the cause might be our use of RDF technologies for producing and consuming message representations, but further investigations are needed to identify implementation bottlenecks. The `STN-A&A` setup performed the worst, which is justified by both the added overheads of the two platforms and the inter-process communication overhead.

The maximum number of sequential contract-net-like interactions that can be completed within the 1 second time limit for each setup is shown in Table 1. Based on these preliminary results, and assuming an optimized implementation of the

STN platform with local BDI agents, we estimate that an STN of 250 agents running on a regular laptop computer should achieve up to 20 sequential contract-net-like interactions in less than 1 second. In the same conditions, we estimate that an STN of 1000 agents should achieve up to 10 sequential contract-net-like interactions in less than 1 second.[19] We expect that a mashup composition requiring 10 contract-net-like interactions would achieve fairly complex functionality. As a comparison, we implemented the realistic application scenario in Sec. 1.1 with only one contract-net interaction.

We conclude that STN-based applications remain responsive when scaling to many devices and for relatively large mashup compositions. Evaluation results, however, also point out the primary challenge for this approach: even though the mashup composition mechanism is straightforward and does not require costly computation, handling and routing many messages in an STN is costly. Pushing every sensor reading to an STN, for instance, might be unwise. Rather we expect that STN communication would be reserved for goal-driven agent interactions and for sharing knowledge (possibly derived from sensory information). Mechanisms to reduce or replace agent communication altogether would be useful (see Sec. 6 for more details).

Further investigation is required to determine the practicality of deploying and running STNs on small devices (e.g., Raspberry Pi). Our current evaluation results, however, seem to indicate that the task of hosting large STNs would be better suited for cloud-based services that can scale to accommodate a growing number of agents.

### Application Flexibility

We consider an IoT application to be flexible if it can adapt to dynamic environments to achieve its design goal. To assess the flexibility of STN-based IoT applications, we implemented the application scenario presented in Sec. 1.1. The developed application is able to wake up its user by composing an online calendar service with sensing and actuation services available at runtime.

#### *Agent and artifact creation*

In our implementation, David's things are modeled as agents. The *A&A container* synchronizes with the *CoRE Resource Directory* to create and maintain a dynamic set of Jason agents for all devices discovered at runtime.

Agents interact with their devices and the *STN platform* via artifacts. The artifact used to interact with the STN platform is created by the *A&A container*, whereas device artifacts are created by the agents themselves.

#### *STN deployment*

Once created, agents are bootstrapped into an STN via preprogrammed generic behavior. Their entry point into the STN is the URI of their owner's user account on the STN platform. Agents register to the platform and declare their owner via the `stn:ownedBy` property (cf. Listing 1), which makes them discoverable via crawling. Agents then implement a behavior to subscribe to all other agents of their owner.

[19] It should be noted, however, that our estimations assume a "hub-centric approach" in which the agents and the STN platform reside on the same machine. Interactions across multiple machines would imply an additional communication overhead.

**Table 2. Beliefs produced and consumed by David's agents.**

| Agent | Produces | Consumes |
|---|---|---|
| Calendar | - | asleep(david) |
| Wristband | asleep(david)[certainty=0.6] | - |
| Mattress cover | asleep(david)[certainty=0.8] | - |
| Curtains | curtains_state(open), curtains_state(closed) | asleep(david), outside_light_level(Level) |
| Lights | lights(on), lights(off) | asleep(david), curtains_state(State), outside_light_level(Level) |
| Light sensor | outside_light_level(Level) | - |

#### *Loosely coupled agent interactions*

Agents exchange messages via the *STN platform* using a common *agent communication language* provided by Jason [3] and a shared *vocabulary* for describing the state of the physical world. For instance, the wristband agent informs other agents that its owner fell asleep by posting to the STN platform the message:

```
tell asleep(david)[certainty(0.6)],
```

which is composed of:

- a *performative* that defines the intention of the communication: `tell` denotes that the wristband agent intends the receivers to believe (that the wristband agent believes) the content of the message to be true;
- a *propositional content*, which is a Jason term that denotes the object of the communication: David is asleep with a certainty of `0.6` (we assume all of David's agents use the same scale of certainty).

Other performatives used in our implementation include:

- `untell`, which denotes that the sender intends the receiver *not* to believe (that the sender believes) the content of the message to be true;
- `achieve`, which is used by the sender to delegate to receivers the goal in the message content;

All knowledge used to implement agent interactions is either standard, or in an easily standardizable form. Agents do not hold any prior knowledge about one another, which enables their independent deployment at runtime.

#### *Flexibility via belief sharing*

Agents use their device artifacts to sense and act upon the IoT environment, which allows them to derive *beliefs* about David and his bedroom. Once deployed and able to interact, agents share belief changes (i.e., when an agent adds/removes beliefs to/from its belief base) via the STN platform to augment one another's capabilities at runtime. The beliefs produced and/or consumed by each agent in the STN are shown in Table 2.

For instance, both the wristband and the mattress cover agents can determine if David is asleep based on sensory information (the mattress cover agent with higher confidence). The calendar agent relies on their beliefs to determine if David is asleep or not. If there is an upcoming event and the calendar agent

holds a belief with certainty above `0.5` that David is asleep, it will attempt to wake him up. If David wakes up, the wristband and mattress cover agents inform the other agents that they no longer believe David is asleep by posting the message `untell asleep(david)`, which enables the calendar agent to verify that it has achieved its goal.

*Flexibility via goal delegation*

David's agents implement the Contract Net Interaction Protocol (see Fig. 1). The calendar agent launches a call for proposals to wake up David by posting to the STN platform the message:

```
tell cfp(<id>, achieve(not asleep(david))),
```

where `<id>` is a unique identifier for the launched interaction. All agents that can trigger alarms to wake up David reply with proposals. The curtains agent, for instance, will only reply if the outside light level is above 100 lux (S.I.), which is the equivalent of an overcast day[20], with the following message:

```
tell propose(<id>, alarm_type(natural_light)).
```

The calendar agent waits for 1 second to receive proposals and then chooses a participant based on David's preferences, which are preconfigured. The delegated agent triggers the alarm and informs the calendar that the action has been performed. The calendar agent, however, remains committed to achieve its goal and awaits for 30 seconds to receive a confirmation via the STN that David woke up (e.g., from the wristband or mattress cover agents). If no confirmation is received, the calendar launches another interaction in a new attempt to wake him up. The calendar agent remains committed to its goal until either an update is published that David woke up, or the scheduled event has passed.

*Flexibility via plan sharing*

Each agent implements plans for handling its specific device artifact. In an alternative implementation of this scenario, David's devices could be represented as a dynamic set of artifacts made available to the calendar agent at runtime, who would than have to learn on-the-fly the plans required to use them. This could be achieved via the `askHow` performative provided by Jason, which enables the calendar agent to ask other calendar agents in his STN for plans to achieve its goals, for instance by posting the message:

```
askHow "+!wakeUp(Owner)",
```

where the content of the message is a triggering event, as defined in Jason, to achieve the goal of waking up someone (`Owner` is an unbound variable).

If none of the agents in the STN know any applicable plans, the calendar agent could use a reasoner to build the plan it needs based on semantic descriptions of its artifacts. The resulting plan could then be shared with the other agents in the STN in future interactions.

The above illustration explores new means through which STN-based IoT applications could achieve flexibility at runtime. It also illustrates how the approach presented in this paper could be used in conjunction with fully automatic composition of service mashups. We leave it as future work to further investigate these concepts.

[20] http://www.engineeringtoolbox.com/light-level-rooms-d_708.html, Accessed: 27.06.2016.

## CONCLUSIONS AND PERSPECTIVES

In this paper, we proposed a decentralized approach to IoT mashup composition that emphasizes both the flexibility and responsiveness of resulting applications. The novelty of our approach is a paradigm shift from monolithic IoT applications to dynamic networks of agents and artifacts, which we call *socio-technical networks (STNs)*. Agents are goal-driven and cooperate with one another to compose IoT mashups at runtime in pursuit of their goals. Furthermore, they use STNs to autonomously discover and interact with one another in a loosely coupled manner, which allows them to be deployed and to evolve independently. Decentralized goal-driven composition and loose coupling are the premises for *application flexibility*. To increase the *responsiveness* of STN-based IoT applications, agents achieve their goals by executing precompiled plans. Evaluation results show that, in the worst-case scenario, the composition overhead grows linearly with the number of agents in the STN, and suggest that applications remain responsive as they scale to many devices and for relatively large mashup compositions.

In the future, we intend to explore new means to enhance the responsiveness of STN-based IoT applications, for instance by studying various topologies and relation management strategies for optimizing the flow of information through the network. Other optimizations could aim to minimize the need for contract-net-like interactions or to remove them altogether. For instance, agents could provide descriptions of the goals they can achieve such that other agents are able to determine in advance their dependencies in a given STN.

In our scenario implementation, we used an ad-hoc solution to represent contextual information about David and his bedroom. There is, however, a large amount of work on context representation and reasoning [19]. We plan to further explore ways in which agents in STNs can be enhanced with context-awareness.

## REFERENCES


1. Michael Blackstock and Rodger Lea. 2012. IoT mashups with the WoTKit. In *Internet of Things (IOT), 2012 3rd International Conference on the*. IEEE, 159–166.
2. Olivier Boissier, Rafael H Bordini, Jomi F Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78, 6 (2013), 747–761.
3. Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming multi-agent systems in AgentSpeak using Jason*. Vol. 8. John Wiley & Sons.
4. Andrei Ciortea, Antoine Zimmermann, Olivier Boissier, and Adina Magda Florea. 2015. 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)*, Vol. 1. IEEE, 461–468.

5. I. Fette and A. Melnikov. 2011. The WebSocket Protocol. RFC 6455 (Proposed Standard). (Dec. 2011). http://www.ietf.org/rfc/rfc6455.txt
6. Foundation for Intelligent Physical Agents. 2002. FIPA Contract Net Interaction Protocol Specification. (2002). http://www.fipa.org/specs/fipa00029/SC00029H.html Accessed: 27.06.2016.
7. Malik Ghallab, Dana Nau, and Paolo Traverso. 2004. *Automated planning: theory & practice*. Elsevier.
8. Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor CM Leung. 2015. Developing IoT applications in the fog: a distributed dataflow approach. In *Internet of Things (IOT), 2015 5th International Conference on the*. IEEE, 155–162.
9. Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. 2009. Towards physical mashups in the Web of Things. In *Networked Sensing Systems (INSS), 2009 Sixth International Conference on*. IEEE, 1–4.
10. Dominique Guinard, Vlad Trifa, and Erik Wilde. 2010. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010*. IEEE, 1–8.
11. Isam Ishaq, David Carels, Girum K Teklemariam, Jeroen Hoebeke, Floris Van den Abeele, Eli De Poorter, Ingrid Moerman, and Piet Demeester. 2013. IETF standardization in the field of the Internet of Things (IoT): a survey. *Journal of Sensor and Actuator Networks* 2, 2 (2013), 235–287.
12. Robert Kleinfeld, Stephan Steglich, Lukasz Radziwonowicz, and Charalampos Doukas. 2014. glue. things: a mashup platform for wiring the Internet of Things with the Internet of Services. In *Proceedings of the 5th International Workshop on Web of Things*. ACM, 16–21.
13. Frank Matthias Kovatsch. 2015. *Scalable Web Technology for the Internet of Things*. Ph.D. Dissertation. Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22398.
14. Matthias Kovatsch, Yassin N Hassan, and Simon Mayer. 2015. Practical semantics for the Internet of Things: Physical states, device mashups, and open questions. In *Internet of Things (IOT), 2015 5th International Conference on the*. IEEE, 54–61.
15. Matthias Kovatsch, Martin Lanter, and Simon Duquennoy. 2012. Actinium: A RESTful runtime container for scriptable Internet of Things applications. In *Internet of Things (IOT), 2012 3rd International Conference on the*. IEEE, 135–142.
16. Matthias Kovatsch, Martin Lanter, and Zach Shelby. 2014. Californium: Scalable cloud services for the Internet of Things with CoAP. In *Internet of Things (IOT), 2014 International Conference on the*. IEEE, 1–6.
17. Simon Mayer, Nadine Inhelder, Ruben Verborgh, Rik Van de Walle, and Friedemann Mattern. 2014. Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning. In *Internet of Things (IOT), 2014 International Conference on the*. IEEE, 61–66.
18. Simon Mayer and David S Karam. 2012. A computational space for the Web of Things. In *Proceedings of the Third International Workshop on the Web of Things*. ACM, 8.
19. Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. Context aware computing for the internet of things: A survey. *IEEE Communications Surveys & Tutorials* 16, 1 (2014), 414–454.
20. Eric Prud'hommeaux and Gavin Carothers. 2014. *RDF 1.1 Turtle - Terse RDF Triple Language, W3C Recommendation 25 February 2014*. W3C Recommendation. World Wide Web Consortium (W3C). http://www.w3.org/TR/2014/REC-turtle-20140225/
21. Alessandro Ricci, Michele Piunti, and Mirko Viroli. 2011. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems* 23, 2 (2011), 158–192.
22. Michael Rietzler, Julia Greim, Marcel Walch, Florian Schaub, Björn Wiedersheim, and Michael Weber. 2013. homeBLOX: introducing process-driven home automation. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*. ACM, 801–808.
23. CoRE Z Shelby and P van der Stok. 2016. CoRE Resource Directory draft-ietf-core-resource-directory-07. *consultant* (2016).
24. Z. Shelby. 2012. Constrained RESTful Environments (CoRE) Link Format. RFC 6690 (Proposed Standard). (Aug. 2012). http://www.ietf.org/rfc/rfc6690.txt
25. Z. Shelby, K. Hartke, and C. Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard). (June 2014). http://www.ietf.org/rfc/rfc7252.txt
26. Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. 2014. Web services composition: A decade's overview. *Information Sciences* 280 (2014), 218–238.
27. Erik Wilde. 2007. Putting things to REST. *School of Information, UC Berkeley* (2007).