# Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications

Matthias Kovatsch
Institute for Pervasive Computing
ETH Zurich
Zurich, Switzerland
Email: kovatsch@inf.ethz.ch

Martin Lanter
Department of Computer Science
ETH Zurich
Zurich, Switzerland
Email: lanterm@student.ethz.ch

Simon Duquennoy
Swedish Institute of Computer Science
Kista, Sweden
Email: simonduq@sics.se

***Abstract*—Programming Internet of Things (IoT) applications is challenging because developers have to be knowledgeable in various technical domains, from low-power networking, over embedded operating systems, to distributed algorithms. Hence, it will be challenging to find enough experts to provide software for the vast number of expected devices, which must also be scalable and particularly safe due to the connection to the physical world. To remedy this situation, we propose an architecture that provides Web-like scripting for low-end devices through Cloud-based application servers and a consistent, RESTful programming model. Our novel runtime container *Actinium (Ac)* exposes scripts, their configuration, and their lifecycle management through a fully RESTful programming interface using the Constrained Application Protocol (CoAP). We endow the JavaScript language with an API for direct interaction with mote-class IoT devices, the *CoapRequest* object, and means to export script data as Web resources. With *Actinium*, applications can be created by simply mashing up resources provided by CoAP servers on devices, other scripts, and classic Web services. We also discuss security considerations and show the suitability of this architecture in terms of performance with our publicly available implementation.**

## I. Introduction

The Internet of Things (IoT) needs new software concepts and architectures that are different from traditional networked embedded devices. The latter have been mostly independent islands, accessed only through application-level gateways, and fulfilling specialized tasks. With the rising presence of lightweight TCP/IP suites [6], [14], this changed and 'things' are becoming directly accessible through the Internet. Programming IoT applications remains unnecessarily difficult, though. Developers have to program different operating systems, focus on platform-dependent issues, and design network interactions. For the IoT to take off, the programming of IoT devices needs to be as easy as scripting a simple Web application.

The Web of Things (WoT) vision proposes to connect things within the IoT using simple, well-defined RESTful interfaces [34]. This is a significant step towards a fully-standardized embedded IoT stack. In our work, we present a system that encompasses constrained devices with only about 10kB of RAM and 100kB of ROM running a Web server directly. Previous WoT solutions usually require either more powerful devices or the Web server being on a gateway. A central design choice is that servers do not include any application-specific code. Instead, they only expose their basic features so that any Internet-connected user or machine can use them for sensing and actuation. [19] We argue that the natural next step to build an IoT upon RESTful devices is to provide IoT developers with the ability to script interactions, independent from Web applications running in the browser.

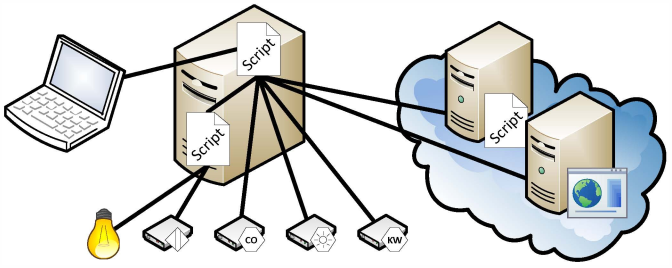

Fig. 1. A script mashes up resources directly from IoT devices as well as other scripts or remote Web services. Our novel runtime container fully complies with the REST architectural style and even performs dynamic installation, updates, monitoring, and removal of scripted applications through RESTful interaction.

Instead of defining a new language, we use the most widespread scripting language in the Web, JavaScript, and enhance it with an API for RESTful interaction with mote-class devices. We propose a novel runtime environment that exposes the scripts themselves through a RESTful interface, making IoT service composition as easy as traditional Web mashups. The runtime container, an IoT application server, may be deployed anywhere with network access: In the Cloud for public applications, or locally on a private server for closed applications. This end-to-end RESTful architecture allows to deploy scripts, to access things, and to call services using the same paradigm, protocols, and, most importantly, programming interface (as illustrated in Figure 1). This paper demonstrates the suitability of Web-like scripting for the IoT. Our solution, *Actinium (Ac)*[1], is fully RESTful in the sense that each entity—devices, applications, and runtime—exposes itself through such an interface.

After reviewing the related work in Section II, we present the design of our architecture in Section III. We include a thorough discussion in Section IV on how authentication and

[1] *Actinium* is publically available at https://github.com/mkovatsc/Actinium

978-1-4673-1346-9/12/$31.00 ©2012 IEEE

privacy can be handled in our model, based on Internet standards. Section V discusses the implementation of our open-source prototype. Finally, Section VI evaluates *Actinium*'s performance in realistic conditions, i.e., with communication flowing between our application server and mote-class devices using our campus IPv6 infrastructure. We show that the overhead of scripting is reasonable, considering the latency of low-power lossy netorks (LLNs) and JavaScript's strength in arithmetic and logic operations.

## II. Related Work

The approaches towards an Internet of Things span various research fields. Here, we summarize in particular how IoT applications are usually developed.

### A. Application Programming

The basis for all device code is an embedded operating system such as *TinyOS*[2] or *Contiki*[3], which is designed for resource-constrained platforms. In the field of wireless sensor networks (WSNs), the application code is traditionally written directly atop the OS. It is often written in the same language as the OS, statically linked to it, and not strictly isolated from it. This makes applications efficient, but also error-prone and complex, as programmers need to know the details of OS and platform. When the software needs to evolve, developers proceed with network-wide full image replacement [16], [28], incremental update [26], or dynamic linking [7].

An interesting alternative is to embed a virtual machine (VM) in the device and deploy applications compiled as bytecode. This approach provides a higher-level of abstraction, provides dynamic loading, software isolation, and minimizes the size of compiled applications. One of the first VMs for sensor networks was *Maté* [21], a framework for domain-specific VMs with mobile code. Later, general-purpose VMs were developed for resource-constrained platforms. *Darjeeling* [4], for instance, supports a large subset of the Java language and even a garbage collector.

Scripting languages raise the level of abstraction even higher, providing the programmer with the ability to batch well-defined basic operations. It is particularly well-suited for WSN programming, as the functionality of each node in the network is built upon a simple set of actions: periodic sensing, alarm triggering, and actuation. This approach increases productivity by making applications self-contained, focused on functionality, and easy to test interactively. [25] On-device script interpretation, as performed by *SensorWare* [3] or *dinam-mite* [10], has comparatively high system requirements. Thus, scripts are usually compiled before sending them to the devices, for instance using Python [1].

Macro-programming is another solution, which aims for programming a network as a whole by providing network-scale abstractions. The early *TinyDB* [22] provided a SQL-like database abstraction for sensor readings of a network. It provided in-network processing where data is aggregated along the hops with convergecast messages. This is, however, not practical for multiple stakeholders of specific readings, as only averages, maximums, etc. can be provided. *Nano-CF* [12] instead is designed for concurrent applications on a WSN infrastructure. It optimizes the execution and traffic of a predefined number of tasks on the motes through *rate harmonized scheduling* and packet aggregation, or concatenation when aggregation is impractical. *Nano-CF* uses code dissemination for its tasks, mostly for homogeneous networks. So does *EcoCast* [30], but instead of a domain-specific language, it uses the Python scripting language to develop applications. The code is then compiled for the platform, incrementally linked, and efficiently patched into the devices.

### B. Application Architectures

Most architectures for IoT-like applications have been ad hoc solutions with TinyOS's and Contiki's custom message formats [24] or proprietary standards such as the ZigBee Cluster Library[4]. Especially macro-programming solutions come with their own specialized protocols, even for media access and routing [12], [22], [30].

With 6LoWPAN [15] and IP stacks for embedded operating systems [6], [14], the system architectures became more standardized and Internet integration of 'things' straightforward. The interoperability, however, often ends beneath the application layer. Despite using UDP messages, prominent deployments such as *ACme* [17] still required application-level gateways to connect to other systems.

For interoperability, the *Devices Profile for Web Services* (DPWS) can bring a Web application standard to low-end devices. With an efficient binding [23] and EXI compression [5], devices can directly process SOAP messages, even on mote-class platforms. Usually, specialized EXI handlers are generated for each application and deployed on the devices.

The *Web of Things* [8], [11], [34] builds upon HTTP for full interoperability at the application layer and adapts the Web's REST architectural style for things. The idea is to provide a simple stateless application-level interface with well-defined semantics. Most WoT deployments still employ application-level gateways, which host the Web servers and wrap the communication towards devices. The Constrained Application Protocol (CoAP) [29], however, brings lightweight UDP-based RESTful interaction to low-end devices [18], [20].

Using REST's uniform interfaces, Web resources can provide the full hardware functionality (e.g., `GET sensor value` or `POST actuation task`) independent from specific applications. Thus, no reprogramming of devices is necessary and multiple applications can leverage the devices at the same time. The idea of a powerful client that executes the main part of the application was already introduced with *Marionette* [33], used in *EcoCast* [30], and pushed further with the *thin server architecture* [19]. Our work combines this idea with Web-like scripting through a novel runtime container.

[2] http://www.tinyos.net/

[3] http://www.contiki-os.net/

[4] http://www.zigbee.org/

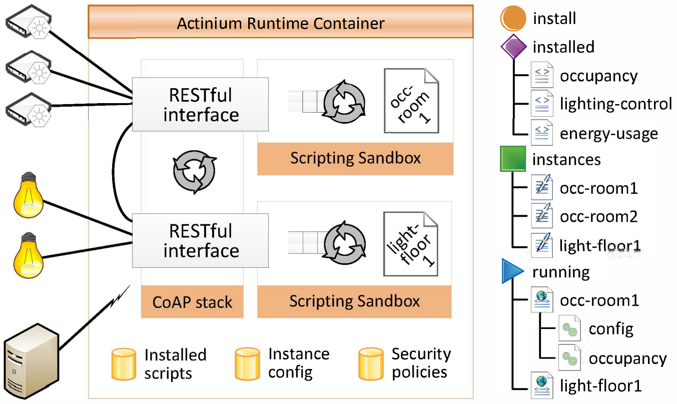

Fig. 2. An overview of our system architecture: Actinium apps are executed in their own threads, isolated in a sandbox. They only communicate through their RESTful interfaces: with IoT devices, other apps, and other servers on the Web. The tree on the right shows the available resource structure when three scripts are installed, two are instantiated (one of them twice), and two instances are running.

## III. Architectural Design

We propose an architecture for networked embedded systems in which applications are realized through scripts running on a server in the Cloud, accessing elementary functionality of IoT devices through their RESTful interfaces. This approach offers a great deal of flexibility and scalability because applications become computer-hosted *apps* rather than embedded software. With *Actinium*, our application server, we extend the WoT approach and advocate a truly end-to-end RESTful approach where not only the devices have RESTful interfaces, but the runtime container itself is RESTful. Our architecture further allows applications to be shared through an ‘appstore,’ i.e., they can easily be uploaded, downloaded, customized, signed, etc. We use the JavaScript scripting language because it is the most prominent language for Web mashups and well-known by many application developers, even by end-users. The architecture, however, is not limited to JavaScript and could also be transferred to other scripting languages.

Although we are aiming for a general RESTful design, in this paper, we focus on the CoAP protocol—the most light-weight protocol for RESTful interaction. Nonetheless, the whole approach does also apply to HTTP-based devices and services where feasible.

### A. Apps as Resources

To implement IoT applications, we use modular *Actinium apps*, which are designed as resources to fully leverage the RESTful paradigm. They can provide their results through GET handlers, accept stimuli by POST, or can be configured via PUT. The central API for this is the `app.root` object, which represents the root resource of an *app*:

```
// a handler for GET requests to "/"
app.root.onget = function(request) {
// that returns CoAP's "2.05 Content" with payload
    request.respond(2.05, "Hello world");
  };
```

We limit *Ac apps* to consist of a single file, which enforces developers to break complex applications down into multiple scripts that have to communicate through their RESTful interfaces. These modules then become reusable by other *apps*, which makes the mashup concept more powerful. Multiple motion sensors, for instance, could be wrapped by one *occupancy app* per room, filtering the sensor data and providing a boolean value as output. A *lighting control app* then combines these occupancy resources with the control resources of the lighting system of each room and automates them for energy savings. Such a chain of *apps* could be continued by a *energy usage app* for example. *Ac apps* can also have sub-resources to provide structure for the exported data:

```
var threshold = 0;
// a sub-resource "/config"
var sub1 = new AppResource("config");
app.root.add(sub1);
// that accepts PUT requests
sub1.onput = function(request) {
// to configure the threshold
    threshold = request.payloadText;
  };
// a sub-resource "/occupancy"
var sub2 = new AppResource("occupancy");
app.root.add(sub2);
sub2.onget = function(request) {
// that returns true or false depending on a given value
    request.respond(2.05, value > threshold ?
      "true" : "false");
  };
```

### B. Runtime Container

IoT scripts require a novel runtime container that is more flexible and easier to use than enterprise application servers. In analogy to the GUI elements of a browser, which are focused on user interaction with event-handlers like `onclick`, our runtime provides IoT-specific elements such as resources that have `onget` and `onpost` handlers, and an `app` object API instead of `window` for facilities such as `app.setTimeout()` or `app.getNanoTime()`. In our design, this container is an *app server* that allows for dynamic installation, updates, and removal of scripts in a RESTful manner: `/install` accepts POSTed scripts, adds and stores them in its resource tree under `/installed`, and reports back the new `Location` via the corrsponding header option. The same *Actinium app* might be required several times, for instance on a per-floor or per-room basis. Thus, we distinguish between installed *apps*, which is the code, and their *instances*, which are created under `/instances` by POSTing an individual configuration to an `/installed/<app>` resource. Scripts can also be stopped or running. In the latter case, they are available under `/running` with their instance name. The full URI of the previous ‘occupancy’ resource would thus look like `coap://app-server.example.com/running/occ-room1/occupancy`

### C. Mashups

The concept of Web mashups is to combine different Web services to provide a service of higher value. Mashups are light-weight applications that are easy to create, provide flexible solutions, and generally leverage the high productivity of scripting. These are properties that are ultimately required for the IoT, where each user is associated to a plethora of heterogeneous devices. Due to the connection to the physical world, powerful applications can already be created by just combining and evaluating sensor or status information, and instantaneously triggering events for actuation or storing the results for data mining.

To mash up devices, *apps* must take the client role. A WebSockets-like API would be an option, but it is for arbitrary data traffic and does not follow a RESTful design. We provide the *CoapRequest* object API, which is designed similar to the XMLHttpRequest object API [31] of AJAX:

```
var req = new CoapRequest();
// request the PIR sensor resource of a mote via CoAP
req.open("GET", "coap://mote1.example.com/sensors/pir",
     false /*synchronous*/);
// with a application/json response
req.setRequestHeader("Accept", "application/json");
req.send();  // blocking
// and log it to the console after send() returns
app.dump(req.responseText);
```

Due to the resource design of *Actinium apps*, they are able not only to mash up services of devices, but also of other *apps* using similar interfaces. The runtime API also supports the normal XMLHttpRequest to include traditional Web services in the mashups. The following snippet contacts a default Contiki border router, which usually hosts an HTTP Web server to list the available routes to connected nodes:

```
var xhr = new XMLHttpRequest();
// GET "/" with a list of all LLN neighbors and routes
xhr.open("GET", "http://br.example.com/", false);
xhr.send();
// and retrieve all LLN node addresses via regular expr.
var addresses = xhr.responseText
                      .match(/[0-9a-z\:]+(?=\/128)/g);
```

Both request objects also support asynchronous communication. Thus, an *app* can also send multiple requests in parallel, which enables interleaving of long-lasting requests to a group of nodes. The responses are then handled by a callback function implementing `onload`. A distinctive feature of CoAP are unreliable requests, which can simplify continuous polling. To choose between *non-confirmable* messages and *confirmables*, an additional boolean is passed to `open` (the default value is `true`, which means *confirmable*, i.e., reliable requests):

```
// define the callback of an existing CoapRequest
req.onload = function() {
    if (this.responseText=="false") switchOffLights();
  };
// and a timeout with a timeout callback
req.timeout = 5000; // in ms
req.ontimeout = function() {
    app.dump("Request timed out!"); // to console
  };
// and send the an asynchronous, non-confirmable request
req.open("GET", "coap://app-server.example.com/
         running/occ-room1/occupancy", // other app
         true /*asynchronous*/, false /*non-confirmable*/)
              ;
req.send(); // non-blocking
// and continue execution immediately
```

A valuable feature of CoAP is *observing resources* [13], which is initiated with the `Observe` header option. These native push notifications can be used similarly to HTTP's chunked transfer (streaming) in AJAX. The `onprogress` callback will inform the *apps* every time an update is received:

```
var req = new CoapRequest();
// define the callback for notifications
req.onprogress = function() {
// unlike XHR,  only contains payload of last message
    update(this.responseText);
  };
// request is DONE, i.e., the observe relationship ended
req.onload = function() {
    app.dump("Observing terminated"); // to console
  };
req.open("GET", "coap://mote1.example.com/sensors/pir",
         true /*asynchronous*/, true /*confirmable*/);
req.setRequestHeader("Observe", 0);
req.send(); // non-blocking
```

## IV. Security Considerations

The IoT is about connecting the virtual to the physical world. As such, it requires security at different levels: First, applications from different providers, running for different users in the same runtime, must not leak sensitive information or be harmful to one another. Second, communication with the devices must be secure, so that sensitive information is authenticated and/or exchanged confidentially. This section reviews how we handle these security issues in our design.

### A. Securing Application Execution

As with operating systems, users have to fully trust the application server. This as prerequisite, trust into individual applications can be relaxed by sandboxing and enforcement of certain policies. Our design provides three mechanisms to this end:

*1) Isolated Apps:* It is crucial that there is no unintended interference between applications. In particular, errors occurring inside an *Actinium app* must not influence the behavior of other *apps*. Our runtime addresses this requirement by holding each *app* within a separate sandbox. The only way for the scripts to communicate with each other is through their RESTful interfaces, either locally or over the network.

*2) Policies:* Keeping *apps* in a sandbox also enables the container to have strong control over them. By default, there are no restrictions in terms of when and how they are allowed to access other resources or to be accessed. An end-user, however, might want to define such restrictions, for instance prohibit activation of the television during night or disallowing *apps* to access local cameras. Our container allows for the definition of such boundaries and guarantees compliance. By providing read-only access to the policies, applications can use defensive programming to avoid relentless trials and crashes.

*3) Monitoring:* Finally, the sandbox wrapper eases the monitoring of the scripts. The runtime records traffic statistics including the amount of transferred data. In addition, it can monitor the CPU time specific threads consume. If the runtime container stresses the CPU, end-users can check which *app* causes the high load. Furthermore, they can check whether the runtime is congested by many incoming requests or actual misbehavior and only stop the script if needed.

### B. Securing Communication

It is important to guarantee that only authorized applications and users interact with a given device, preventing malicious users from getting unauthorized information, or worse, triggering unwanted actions with unpredictable impact on the physical world. Following the open standards, we base our security architecture on the DTLS protocol [27], an adaptation of TLS for UDP providing CoAP with end-to-end integrity, authentication, and confidentiality. There are two aspects for integrating this security model into our architecture:

*1) Authenticating Applications:* Central questions for application authentication are when, how, and to whom keys and certificates should be distributed. We argue that the traditional model of the Internet, where applications (e.g., Web sites or smart phone apps) are only signed by the providers, is not suitable for the IoT. Usually, authenticated applications may access any device and users do not want to grant access to their things based on the application designers' choices. Thus, we propose a model in which the users sign each *configured instance* of an application instead of providers signing the distributable code. Each user has a personal certificate that is uploaded to the owned devices. This upload can be secured using a pre-shared secret shipped by the manufacturer with the device. For each *Ac app* that is deployed by the user, a key pair is generated to produce an *app certificate*, which is signed by the user. *Actinium* can then authenticate the script to the device through its certificate when establishing the DTLS session. The device accepts requests by *apps* only if its certificate is signed by one of the authorized owners.

*2) Providing Data Integrity and Confidentiality:* Another concern in the IoT is to check the integrity of data originating from devices and to guarantee confidentiality while transporting it. This can be done following the same scheme as on the Internet, i.e., by distributing a certificate and a private key to each device. DTLS, as TLS, allows authentication of both parties during the session establishment. By using this feature, an application can be guaranteed about the identity of the data source, and can send and receive data confidentially through any network.

Experiences from TLS, however, show that proper, user-friendly tool support is required to make a certificate-based security model work. Thus, corresponding mechanisms should be provided with a runtime container.

## V. Implementation

To the best of our knowledge, there are five comprehensive open-source implementations of CoAP available: *libcoap*[5] and *Erbium*[6] written in C, *evcoap*[7] in C++, *Californium*[8] and *JCoAP*[9] in Java, and *Copper*[10] in JavaScript. The latter sounds promising for scripting, however, it only implements the client role and is bound to Mozilla's XPCOM API, as it is a Firefox add-on. From the remaining options, Californium implements all required features, such as *observing* [13] and *blockwise transfers* [2], and provides a convenient framework to implement server functionality. Thus, we chose Californium together with Mozilla's Java-based *Rhino*[11] JavaScript engine to create *Actinium*, our IoT application runtime container.

The current version of Californium has a 'single thread' model, though, which is impractical for autonomous *apps* that can issue blocking requests themselves. Thus, we extended the request dispatcher with separate event queues for each running *app*, which are executed in their own threads. These queues are also used for JavaScript's event-driven execution model, i.e., they also hold the timeout and interval events.

AJAX's XMLHttpRequest is not part of ECMAScript [9] and hence not supported by Rhino. We used the *E4XUtils* extension library for ECMAScript for XML (E4X) available from IBM[12] to include this API. Our *CoapRequest* object API is backed up by custom Java code that wraps Californium's client functionality. The new working draft of the *XMLHttpRequest Level 2* specification states that "some implementations support protocols in addition to HTTP and HTTPS" [32]. As the functionality, however, slightly differs from CoAP and the name of the object would become even more confusing, we decided for separate APIs. In future work, we plan to integrate a unifying API that better suites the REST abstraction and is free of the XML legacy.

As complex applications shall be built by mashing up other *apps*, there is no container format such as an JAR-like archive. Each *app* is a plaintext script, which is persisted in a single file and loaded into a wrapper object for execution. The wrappers also implement the sandboxing described in the previous section.

The secure communication is the only part currently not implemented by *Actinium*, as at the time of writing, no implementation of the *coaps* scheme nor DTLS 1.2 was available. We therefore leave the security evaluation to future work.

## VI. Evaluation

### A. Setup

In our experiments, the *app server* is running on a 64-bit Windows 7 Workstation with an Intel Core2 Q9400 @2.66GHz, 8GB RAM, and JavaSE-1.6. The network configuration, which utilizes our campus IPv6 infrastructure, is depicted in Figure 3. To be able to easily sniff the transit traffic, the border router is connected to a Laptop for the experiments.

For the LLN, we use a pre-release of Contiki 2.6[13] with the *rpl-border-router* running on a *Tmote Sky*[14] with DMA enabled for the serial line and the *Erbium* [18] server on *Econotags*[15]. As we focus on the *app server* performance, we configured the LLN with a best case scenario for applications: no radio duty cycling for minimal latency. In a real-world deployment, latency will have to be traded for battery lifetime. An energy evaluation of CoAP over a radio duty cycling layer

[5] http://sourceforge.net/projects/libcoap/

[6] http://contiki.git.sourceforge.net/git/gitweb-index.cgi

[7] https://github.com/koanlogic/webthings/tree/master/bridge/sw/lib/evcoap

[8] https://github.com/mkovatsc/Californium

[9] http://code.google.com/p/jcoap/

[10] https://github.com/mkovatsc/Copper

[11] http://www.mozilla.org/rhino/

[12] http://www.ibm.com/developerworks/webservices/library/ws-ajax1/

[13] http://sourceforge.net/projects/contiki/

[14] http://www.sentilla.com/files/pdf/eol/tmote-sky-datasheet.pdf

[15] http://redwirellc.com/store/node/1

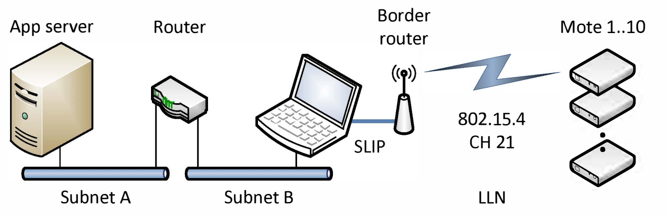

Fig. 3. The border router has a distance of three hops to the app server. As the bottleneck is the LLN, we configured it with a best case scenario for applications: a single-hop star topology without radio duty cycling.

can be found in [18]. Yet, the LLN underlies realistic Wi-Fi interference, as we used 802.15.4 channel 21 with several surrounding access points on channels 1, 5, 9, and 13.

### B. *Latency Baseline*

We evaluated the latency overhead that is introduced by the Rhino scripting environment and our *CoapRequest* abstraction. For that, we compare the round-trip time (RTT) of a JavaScript request over a native request in Java and the network latency measured with Windows's ping tool. Each measurement was executed 1000 times with a IPv6 packet size of 80 bytes. The results summarized in Table I only show the latency for the *apps* in client role, but the resource handlers for the sever role have the same properties through reciprocity.

The average CoAP RTT when including a single LLN hop behind the border router is already 46ms. Thus, the overhead of 1.282ms added by the scripting environment is negligible. Especially when considering that to gain a longer lifetime, battery-powered IoT devices will employ radio duty cycling, which increases the underlying network latency further. LLNs usually aim for an idle duty cycle (i.e., idle listening only without transmissions or interference) well below 1%. ContikiMAC, for instance, achieves 0.6% with a channel check rate of 8Hz, which can add an extra of up to 2·125ms=250ms to the RTT for a single hop.

### C. *REST Handler Performance*

With a negligible network overhead for the scripting abstractions, only the performance of JavaScript could become a showstopper. So, we evaluated the execution times of *Actinium*'s REST handlers with measurements that directly continue from the baselines identified in the last sub-section. We compare the Rhino JavaScript runtime of *Ac* to a native Californium handler in Java and the runtime of node.js[16], a platform that enables server-sided JavaScript for HTTP-based applications leveraging Chrome's V8 JavaScript engine.

To assess different aspects, we used three different benchmarks that are also included in the *Actinium* repository. Each one is implemented as request handler and measures the execution time only, i.e., without runtime start-up and the like. The input parameters are chosen so that the different complexity classes ($O(n^2)$, $O(n \cdot \log(n))$, and $O(n)$, resp.) produce comparable execution times of up to 1.5 seconds.

TABLE I
BASELINE TIMINGS

| | *Minimum* | *Maximum* | *Average* | *Overhead* |
|---|---|---|---|---|
| Ping to BR | 16ms | 62ms | 37ms | — |
| Ping to node | 32ms | 77ms | 46ms | +9ms |
| CoAP/Cf RTT | 34.173ms | 77.587ms | 47.650ms | +2ms |
| Actinium RTT | 32.901ms | 97.088ms | 48.932ms | +1ms |

The timings were measured over 4 hops (1 LLN hop) with 1000 requests per measured system (note that Windows ping only provides 1ms resolution). The smaller minimum for the Actinium RTT is caused by random effects along the stack such as the IEEE 802.15.4 CSMA backoff and the Contiki scheduler.

*1) Fibonacci Benchmark:* The Recursive Fibonacci algorithm causes a large number of function calls and thus shows how efficient the runtime systems manage deeply nested function calls. Figure 4 shows the outcome as assumed: Java is 3.13 times faster than node.js, which is already 3.76 times faster than Rhino because of its efficient C++-based V8 runtime.

*2) Quicksort Benchmark:* The next benchmark sorts an array of double-precision floating point numbers using the Quicksort algorithm, which shows how efficient the runtime systems handle memory access. Unlike the Fibonacci benchmark, the performance factors are not constant over the input parameters. Compared to Java's average speed-up of 18.2, both JavaScript runtimes degrade with increasing array sizes. node.js scales a little worse, but on average it still performs 7.13 times better than Rhino (cf. Figure 5).

*3) Newton Square Roots Benchmark:* Newton's Square Root is a fixed-point algorithm that iteratively computes the square root for a number. Since the result does not matter, we arbitrarily define eight iterations and vary the number of calculated roots. With this algorithm, we compare how efficiently the runtime systems executes arithmetic operations. In Figure 6, the speed-up factors of 3.25 and 4.14 for Java and node.js, respectively, are close and clearly show the strength of scripting for this kind of computation.

On the one hand, the *app server* evaluation shows that Rhino is not a high-performance runtime. On the other hand, arithmetic and logical operations perform comparatively well in JavaScript. This shows that scripting is well suited for the targeted use-case, where RESTful device resources are mashed up to create IoT applications. Memory-intensive tasks like persistent logging or data mining can be outsourced to standalone services with a RESTful API (e.g., a RESTful database). With the new *InvokeDynamic* bytecode instruction in Java 7, the JVM also provides better support for dynamically typed languages.[17] The next version of Rhino is thus expected to provide a performance similar to Chrome's V8.

### D. *Concurrent Apps*

*Actinium* supports multiple apps running and communicating at the same time through message multiplexing. As spawning new threads is not a problem for a back-end system,

[16] http://nodejs.org/

[17] http://jcp.org/en/jsr/detail?id=292

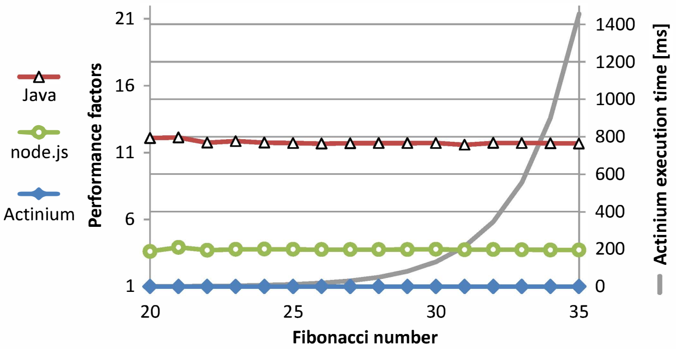

Fig. 4. Fibonacci over different function parameters: The left y-axis shows the performance factors between the three runtimes. Rhino performs on average 11.8 times slower than Java and about 3.8 times slower than node.js. The y-axis on the right indicates Actinium's absolute timings on our test system. We only show these, as the curves look qualitatively the same for all three runtimes.

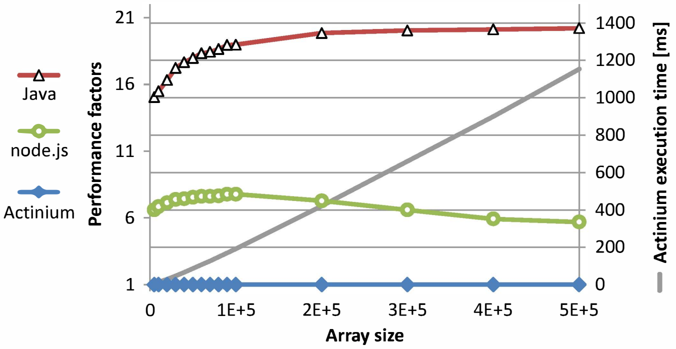

Fig. 5. Quicksort over varying array sizes: For memory-access-intensive tasks, Java performs best with the largest overall speed-up. For this benchmark, the speed-up factors vary and Java even gains performance while node.js slightly degrades when the arrays become very large.

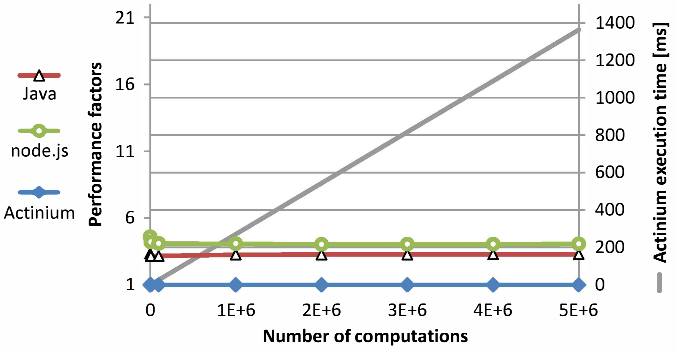

Fig. 6. Newton over a growing set of processed numbers: As Newton's method has a steady linear growth rate, less measurements were taken. An interesting result is to confirm that JavaScript has its strengths in pure computations and node.js even outperforms Java.

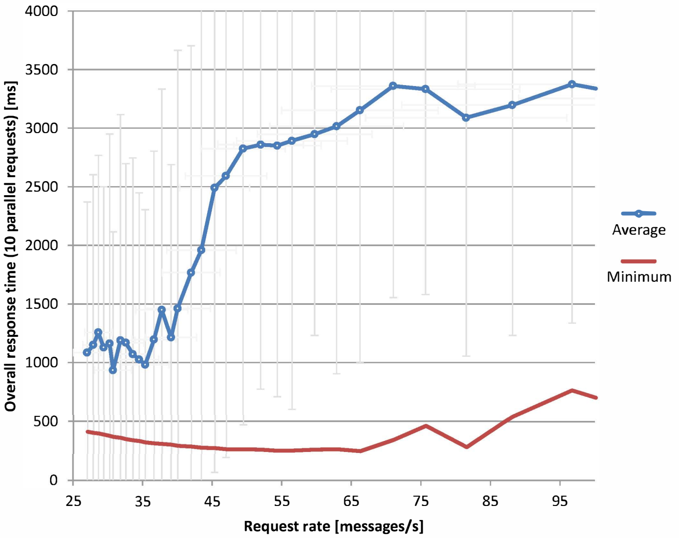

Fig. 7. The overall response time of 10 responses for 10 requests sent with different rates. The graph also shows the measured minima, in which case no interference, thus no restransmissions, occurred. Note that the high average and standard deviation is caused by CoAP's binary exponential backoff for retransmissions, which starts with a random value between 2 and 3 seconds by protocol default.

we have to investigate the network traffic to reason about concurrently running scripts. On our test system, the rate of sending 80-byte requests from the JavaScript runtime converges to about 2500 messages per second. The bottleneck lies of course in the destination LLN, which can become congested with too many messages. We conducted an experiment where ten asynchronous requests are sent to ten different nodes in one LLN in parallel. We recorded the overall response time, i.e., the RTT between the first outgoing request and the last incoming response, and varied the outgoing rate of messages. For this, we used an *Ac app*[18] that simply calls `app.sleep(delay)` between sending the requests and measures the timing with `app.getNanoTime()`. The experiment was repeated about 500 times, whereas we filtered 1.8% of the runs because they did not complete within our overall request timeout of 20s.

Figure 7 shows a drastic increase in latency around 45 requests/s. This is where the LLN becomes congested and link-layer retransmissions exceed the channel capacity. Thus, an application-layer retransmission is required, which in a default CoAP configuration occurs after two to three seconds and is repeated after twice the previous interval until four retransmissions. Without interference, rates beyond the 45 requests/s mark can also achieve overall RTTs below 500ms. Also note that with knowledge of the LLN and the applications, a lower average can be produced by tweaking the CoAP parameters.

We used these results to implement a rate limitation layer for Californium. This only covers the scenario of one *app server* for a single, known LLN. In a real-world deployment, the traffic shaping should be integrated into the border routers, as the channel properties apply per LLN and *Actinium* might use multiple deployments. Furthermore, caching should be employed at the runtime container and at the border of each LLN. Caching has the same result as Nano-CF's packet aggregation and concatenation [12], reducing the traffic in the LLN, but better decouples applications and infrastructure.

[18] The app has 21 Logical Source Lines of Code (LLOC) for the measurement, and 37 LLOC in total including RESTful facilities to retrieve the node addresses from the border router, set the delay, and start the measurement.

## VII. Conclusion

The goal of this paper is to make programming of Internet of Things applications significantly easier. We propose an architecture for the IoT that unifies the Web of Things idea with the requirements of constrained mote-class devices. Our fully RESTful runtime container *Actinium (Ac)* allows for dynamic installation, update, and removal of scripts. These *apps* are modelled as resources themselves and can provide parameters, status information, and results through RESTful interfaces. Complex applications are implemented by mashing up such resources, provided directly by 'things,' other *apps*, or classic Web services. Security is provided through traditional Internet standards, but with a paradigm change in how applications are signed. *Ac* enables the integration of low-end sensors and actuators into the Internet, whereas we see the IoT more as an Internet *with* Things rather than an Internet *among* Things.

The evaluation of our working prototype shows that the performance of the runtime container is relaxed by the latency of low-power networks, which connect 'things' to the Internet. As the scale of the IoT is expected to be immense, though, the runtime must handle several applications, each orchestrating many devices, potentially in multiple networks. For computational tasks, JavaScript can outperform a native Java implementation. Thus, scripting is a viable solution for the mashup programming model, which connects different REST resources through logic and arithmetic operations to provide services of higher value.

Our proposed architecture can kindle end-user programming for devices in a new way, as scripting even addresses beginners. Our security model based on user-signed *apps* allow users to build trusted IoT applications for their 'things.' An adoption of our *CoapRequest* object by Web browsers could fully integrate things into the Web. For applications that require more user interaction, the runtime container could also be a desktop widget, powered by a default browser engine. A continuation of our design will focus on a generalization of an object API for RESTful scripting, which combines CoAP and HTTP, or any other future RESTful protocol.

## Acknowledgment

This work was partly supported by CONET, the Cooperating Objects Network of Excellence, under EU-FP7 contract number FP7-2007-2-224053, as well as SSF through the Promos project.

## References

[1] SNAP based Wireless Networks. Technical Report SNAP Whitepaper, Synapse Wireless, 2008.

[2] C. Bormann and Z. Shelby. Blockwise transfers in CoAP. draft-ietf-core-block-08, 2012.

[3] A. Boulis, C. Han, and M. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proc. MobiSys*, San Francisco, CA, USA, 2003.

[4] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich VM for the resource poor. In *Proc. SenSys*, Berkeley, California, 2009.

[5] A. Castellani, M. Gheda, N. Bui, M. Rossi, and M. Zorzi. Web Services for the Internet of Things through CoAP and EXI. In *Proc. ICC*, Kyoto, Japan, 2011.

[6] A. Dunkels. Full TCP/IP for 8-bit Architectures. In *Proc. MobiSys*, San Francisco, CA, USA, 2003.

[7] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks. In *Proc. SenSys*, Boulder, Colorado, USA, 2006.

[8] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. The Web of Things: Interconnecting Devices with High Usability and Performance. In *Proc. ICESS*, HangZhou, Zhejiang, China, May 2009.

[9] ECMA Int. ECMAScript Language Specification 5.1. ECMA-262, 2011.

[10] D. Gordon, M. A. Neumann, and M. Beigl. Demo Abstract: Program Your Reality with dinam-mite. In *Proc. Pervasive*, San Francisco, CA, 2011.

[11] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In *Proc. IoT*, Tokyo, Japan, 2010.

[12] V. Gupta, J. Kim, A. Pandya, K. Lakshmanan, R. Rajkumar, and E. Tovar. Nano-CF: A Coordination Framework for Macro-Programming in Wireless Sensor Networks. In *Proc. SECON*, Salt Lake City, UT, USA, 2011.

[13] K. Hartke. Observing Resources in CoAP. draft-ietf-core-observe-05, 2012.

[14] J. Hui and D. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proc. SenSys*, Raleigh, NC, USA, 2008.

[15] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. RFC6282, 2011.

[16] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proc. SenSys*, Baltimore, MD, USA, 2004.

[17] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and Implementation of a High-Fidelity AC Metering Network. In *Proc. IPSN*, Washington, DC, USA, 2009.

[18] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proc. MASS*, Valencia, Spain, 2011.

[19] M. Kovatsch, S. Mayer, and B. Ostermaier. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *Proc. IMIS*, Palermo, Italy, 2012.

[20] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proc.IP+SN*, Chicago, IL, USA, 2011.

[21] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proc. ASPLOS-X*, San Jose, CA, USA, 2002.

[22] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *Trans. Database Systems*, 30(1):122–173, 2005.

[23] G. Moritz, F. Golatowski, and D. Timmermann. A Lightweight SOAP over CoAP Transport Binding for Resource Constraint Networks. In *Proc. MASS*, Valencia, Spain, 2011.

[24] L. Mottola and G. Picco. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *Computing Surveys*, 43(4), 2011.

[25] J. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

[26] R. K. Panta, S. Bagchi, and S. P. Midkiff. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation. In *Proc. USENIX*, San Diego, CA, USA, 2009.

[27] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC4347, 2006.

[28] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi. SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes. *Trans. Mobile Computing*, 9(12):1749–1765, 2010.

[29] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). draft-ietf-core-coap-11, 2012.

[30] Y.-H. Tu, Y.-C. Li, T.-C. Chien, and P. H. Chou. EcoCast: Interactive, Object-Oriented Macroprogramming for Networks of Ultra-Compact Wireless Sensor Nodes. In *Proc. IPSN*, Chicago, IL, USA, 2011.

[31] W3C. XMLHttpRequest. Editor's Draft 4 May 2012.

[32] W3C. XMLHttpRequest Level 2. W3C Working Draft 17 Jan 2012.

[33] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *Proc. IPSN*, Nashville, TN, USA, 2006.

[34] E. Wilde. Putting Things to REST. Technical Report 2007-015, School of Information, UC Berkeley, Berkeley, CA, USA, 2007.