ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

RESEARCH GROUP FOR
Distributed Systems

Bachelor's Thesis

at the Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

Actinium: An App Server for the Smart Home

by Martin Lanter

Spring 2012

ETH student ID: 08-928-368
E-mail address: lanterm@student.ethz.ch

Supervisiors: Dipl.-Ing. Matthias Kovatsch
Prof. Dr. Friedemann Mattern

Date of submission: 18 April 2012

Affidavit

I hereby declare that this bachelor thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

This thesis has not yet been presented to any examination authority, neither in this form nor in a modified version.

By signing this statement, I affirm that I have read the information notice on plagiarism, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Information notice on plagiarism:

`http://www.ethz.ch/students/semester/plagiarism_s_en.pdf`

Zurich, 18 April 2012

Martin Lanter

Abstract

We present the Actinium (Ac) App-server for Californium, a server design for apps written in JavaScript. The targeted use-case is the smart home environment, but the concept can also be ported to business and industry scenarios. Actinium extends the Californium (Cf) CoAP framework by integrating the Mozilla Rhino engine to provide support for JavaScript apps. Apps can dynamically be installed, removed, and updated at runtime and can export their configuration and output as CoAP resources. Furthermore, we propose the CoAPRequest API, an interface for JavaScript apps to communicate asynchronously and synchronously with other CoAP resources. The CoAPRequest API is designed similarly to AJAX's XMLHttpRequest API. Actinium monitors all the apps and provides information about them, such as CPU usage, message throughput and more. Although performing about factor ten slower than native implementations, the application logic is executed fast enough to orchestrate devices in a low-power network.

Keywords

Internet of Things, Smart Apps, Californium, CoAP, JavaScript

Contents

Abstract **v**

Objectives **ix**

1 Introduction **1**

2 Design Overview **3**
- 2.1 Configurations . . . 3
- 2.2 Resource Layout . . . 5
- 2.3 Installation process . . . 7
- 2.4 Isolation of apps from the underlying system . . . 7

3 Implementation **11**
- 3.1 CoAPRequest . . . 11
- 3.2 Encapsulated commands . . . 12
- 3.3 The stats resource . . . 13

4 Performance Evaluation **15**
- 4.1 Benchmark Methodology . . . 15
- 4.2 Results . . . 16
- 4.3 Outlook . . . 19

5 Conclusion **21**

A JavaScript and CoAPRequest Examples **25**

Bibliography **29**

Objectives

In a first step, the student shall integrate Rhino into Californium. This allows Californium to interpret JavaScript scripts. The apps shall be (un-)loadable at runtime and export their configuration and output (e.g., status or intermediate results) as CoAP resources.

Next, the student shall develop an API that allows JavaScript apps to use Californium's CoAP stack. Server aspects shall be mapped to CoAP resources, which can be defined in the script together with their handler functions. Client functionality shall be provided through an object similar to AJAX's XMLHttpRequestObject that provides the same functionality for CoAP.

Finally, the student shall evaluate the app server. On the one hand, the performance must be benchmarked: All application logic is stripped from the devices, so that their processing power is unutilized. This reduces complexity and hardware requirements, but introduces communication overhead. On the other hand, the design space could be limited. For examination, the student shall create different application kernels that are typical for the Internet of Things and wireless sensor networks (e.g., data collection, actuation, and control loops).

1 Introduction

Today's homes encompass a notably large collection of electronic devices and appliances. While the Internet is unabatedly growing and extending to the physical world, all kinds of appliances are being incorporated into it. In the future, freezers, light switches, and thermostats are expected to be part of this Internet of Things. A household is going to be a collection of smart appliances, interconnected and interacting with each other. Together with each appliance, an app can be shipped that controls the appliance and augments its functionality. For instance, pricing information from the smart grid could adjust the temperature of a freezer or an app could monitor the total load of our power outlets to comprehend energy costs. Powerful appliances, such as the router or TV set, will be able to host the apps. New apps can be installed and old apps can be deleted or updated with the most recent version. Apps can work alone, together or build upon each other.

We assume a Web-like infrastructure of devices that export their functionality through RESTful Web Services [2] [6]. Through these interfaces, we are able to control the apps and retrieve data from them. For instance, we will adjust the desired room temperature not physically with a button on certain devices, like heaters or radiators, but change the desired value on a virtual app that is responsible for this matter. The app then manages the available heaters and radiators to achieve the desired room temperature. Just as well, instead of watching the display of a thermostat, we retrieve the current temperature from this app, for instance through a desktop widget or smartphone app. Moreover, other apps that depend on the current room temperature might retrieve it from the temperature app in the same way or register as observer and be notified when the temperature changes.

To connect apps and devices we require a communication protocol. Kovatsch [4] suggests the Constrained Application Protocol (CoAP) [10], a RESTful protocol similar to HTTP that operates over UDP instead of TCP. Kovatsch [3] argues that smart devices and apps will be an extension to the Web and therefore are to use Web technologies like CoAP for heterogeneous systems. CoAP combines well-proven concepts from the Internet and allows a scalable, resource-oriented architecture [4] [7]. Obersteg and Pauli [9] proposed Californium, a modular CoAP supporting framework, written in Java. Resources that serve as "heavy-weight" apps can easily be written with Java and added to the resource tree. However, installing, removing and updating these resources cannot yet be done dynamically but require a restart of the framework.

Maintainability of apps is a major factor for end-users. People expect easy access to apps and strong control over which apps are installed and removed. To this end, the

underlying system must provide straightforward and uncomplicated means to manage its apps [1]. People will not tolerate to restart the whole system just to add, remove or update a single app.

Furthermore an end-user will expect his devices to support app-based control and the availability of a broad variety of apps for those devices. Thus, the industry must be able to provide solutions for these needs. To enhance the productivity of development teams, the program language to construct apps must be as simple as possible, powerful though. Scripting languages like JavaScript serve this purpose very well, are easy to learn and already well known by many developers. With support for scripting, even informed end-users could customize apps or even create them by themselves. Appropriate APIs enable developer to integrate complex tasks like communication with other apps or sources from the internet in a simple, reliable and intuitive way.

To meet these goals, we propose an app-server design that allows to dynamically installing, removing and updating apps. We implemented our design in Actinium (Ac) App-server for Californium. Actinium supports scripting apps by integrating Mozilla's JavaScript engine Rhino and arms it with a powerful API, called CoAPRequest [8], to communicate with other resources. Actinium enables full control for end-users by monitoring apps in detail and recording incomming messages.

2 Design Overview

In Actinium, it is possible to use the same app more than once. Reusing the code of an app not only saves storage space, but also allows us to update an app's code and automatically also update all its instances. If there is an app that observes the power consumption of one or more devices for example, we might create one app-instance of this app that observes all devices on the first floor, and another independent app-instance that observes all devices from the second floor. There is no artificial limitation on how many instances of an app are created. App-instances are independent from each other, the only similarity being, that they execute the same program code.

Actinium exports itself as a resource with subresources. It provides separate subresources for installing, deleting, updating, monitoring and configuring app-instances. In addition, it contains the app instances itself, which are resources, containing their own subresources. Actinium, apps and app-instances, all have different properties. We first describe what kinds of properties Actinium has to manage and then give an overview over the structure of the corresponding resources.

2.1 Configurations

There are several classes of properties that an end-user or a developer might want to configure. We call a set of properties a **configuration**. We identify four classes of properties that influence the behavior of the server, apps or app-instances:

Server properties: Properties that belong to the whole server are called server properties, e.g., the path to the directory on the disk, where the scripts of installed apps are saved to, is a server property.

App properties: Properties that are relevant for an app in general and all its instances, e.g., that an app is an app written in JavaScript and therefore has to be interpreted by a JavaScript engine is an app property. So far, this and the app's name are the only app properties that Actinium uses and are therefore integrated into the instance properties. However, it is possible that more such properties occur in the future and should be treated apart from instance properties.

Instance properties: Properties that control a single app-instance are termed instance properties. For example, the name of the app-instance, which also is the instance's resource's name, is an instance property. Just as well, whether the specific instance is started, stopped or restarting at the moment, is an instance property. Instance properties are restrictions and boundaries for an app-instance. They control the behavior of an app-instance and not vice-versa. If we introduced policies at some future point, e.g. not allowing an app-instance to run during the night or forbid to access some specific other resources, we would implement such policies as instance properties.

It is possible to define (start-) parameters for an app-instance. In our example from above with the power consumption observing app, we define a parameter that tells the app code, which floor's devices it is supposed to observe. Apps, meaning the code, can read but not write these properties. It is a crucial design decision to not let app-instances modify their properties themselves. Otherwise, if an app instance managed its instance properties itself, it might just refuse to accept property updates. Be it because of a careless implementation by the developer of an app or because of an intended denial of cooperation, an app might just not initiate its own shutdown, for example. Instead, there is one separate resource per app-instance that holds the corresponding set of instance properties (configuration). Actinium provides a specific parent resource that contains all these app-instances' configuration resources. To modify instance properties, we send requests to the responsible resource which then changes the values, notifies potential subscribers and then takes actions according to the new values. For example, when we send the resource a request to stop the corresponding app-instance and the resource will initiate its shutdown.

Currently, Actinium uses 12 predefined instance properties. See table 2.1.

App internal properties: Properties that are part of the app's program code are termed internal properties. These are the properties that an app instance manages itself and possibly exports with its subresources. For example, the sampling interval that an app uses to update its values is an internal property. The app might provide a subresource which represents that property and to which a request can be sent to either retrieve or adjust it.

Instance properties		
Property	**Type**	**Description**
app	String	Name of the app.
dir_path	String	Path to the directory that contains the app.
type	String	Type of the app (e.g. JavaScript).
name	String	Name of the app-instance.
resource_title	String	Title of the CoAP resource.
resource_type	String	Type of the CoAP resource.
start_on_startup	true/false	Whether the app-instance should automatically start, when Activium starts up or not.
allow_output	true/false	Whether the app-instance is allowed to print to the standard output stream.
allow_error_output	true/false	Whether the app-instance is allowed to print to the standard error output stream.
enable_request	true/false	Whether requests for the app-instance get delivered or not.
running	start/stop/restart	Running status of the app-instance.
availability	available/ unavailable	Whether this app-instance is available or has been deleted and is going to be cleaned up.

Table 2.1: Predefined instance properties of an app-instance.

2.2 Resource Layout

The resource structure of Actinium has to represent several entities. First, singleton resources that represent global data like the server properties or the statistics resource. Second, ports for installing, deleting and updating apps. Finally, a tree with resources for apps, their app-instances and their instance properties (configuration). An app-instance must have the same name as the corresponding resource that holds its configuration.

Actinium's functionality is split up into five different resources:

/install: The install-resource is the reception point for new apps and the parent resource of all installed apps. See chapter 2.3 Installation Process.

/config: This resource represents the server properties.

/stats: The stats-resource monitors the app's instances. See chapter 3.3 Stats Resource.

/apps/appconfigs: The appconfigs-resource holds one subresource per app-instance. Each such subresource contains the instance-properties for its corresponding app-instance.

/apps/running: The running-resource holds the running app-instance's subresources. There is exactly one root-resource per app-instance which can contain further subresources. If an app-instance shuts down, it will be removed from the running-resource.

In this section, we discuss the different resource layouts we have evaluated and compare them to each other. Let an *app* be the code of an app, that has been sent to Actinium. Imagine we had n apps app_1, app_2, $\ldots$, app_n. Of each app_k we had n_k instances $instance_{k,1}$, $instance_{k,2}$, $\ldots instance_{k,n_k}$ and n_k configurations $config_{k,1}$, $config_{k,2}$, $\ldots$ $config_{k,n_k}$.

We have evaluated three different resource layouts of which we have chosen the first one.

- Grouped by resource type layout, see figure 2.1 a)
- Grouped configs layout, see figure 2.1 b)
- Grouped apps layout, see figure 2.1 c)

With the chosen resource layout, we group the different types of resources (apps, configs and instances) together under a common subresource per type. Thus, the install-resource contains all n apps, the appconfigs-resource contains all configurations $config_{k,i}$ and the running-resource contains all $instance_{k,i}$ that are running at the moment.

In the second layout the configurations $config_{k,i}$ are placed right under their corresponding app_k. Thus splitting up the group of configurations and distributing them over the installed apps. As a result, the controlling entities (the code and instance properties) of an app-instance lie close to each other. The running app-instances $instace_{k,i}$ remain grouped together as subresources of the running-resource.

In both layouts, there is a clear separation between the installed apps and their running app-instances since they are subresources of two different parents. Hence, the code of an app can be maintained independently from its app-instances. Since all instances are siblings, they must have different names. This could be seen as a disadvantageous restriction of names or as advantageous guarantee for an absence of name conflicts between app-instances.

A disadvantage of the grouped configs layout is the mapping between a running app-instance and its corresponding configuration. The set of app-instances is not sorted in any way (unlike in the diagram). For a user, who knows the name *instancename* of an app-instance, this is the only hint he has to find the corresponding configuration. However, these configurations are distributed over all n installed apps. To find the correct URI /install/*appname*/*instancename*, he has to search through all the apps until he finds the installed app *appname* that contains the configuration $instancename$. In contrast, in the first layout, if we know the name *instancename*, we immediately know the URL for the corresponding configuration: /install/appconfigs/$instancename$.

In the third resource layout the app-instances are split up and distributed over the apps as well. Thus, each app app_k has two subresources: running for its running app-instances $instance_{k,i}$ and appconfigs for its app-instances' configurations $config_{k,i}$. All resources that somehow relate to the same app structure a separate tree.

Since neither the app-instances nor the configurations of different apps are siblings, we have more freedom for names. In the grouped apps layout, app-instances of different apps are allowed to have the same name. We also do not have a problem, finding the URI of a configuration

corresponding to an arbitrary app-instance. However, a running app-instance is a grand-child of the resource that represents the app and its code. Hence, maintainability of an app's code is not independent of its app-instances.

A further advantage of the chosen layout, that groups resources by their types, is the extendibility of the resources for installed apps. In the second and third layout, these resources app_k have subresources, where in the first they do not. Therefore, we can extend the model of an app in the future. In particular, this might come in handy, if we end up integrating broader support for app properties (which are not instance properties) at some future point.

In conclusion, we decided for the tradeoff of the grouped by resource type layout, because it provides a consequent grouping of the different resource types, independence of app-instances and their code, easy URI mapping and extensibility.

2.3 Installation process

The installation process starts at the install-resource. It expects a POST request with the app's JavaScript code as payload. The request has to be addressed to the install resource's URI combined with a query that contains the name of the new app. To add a new app called *myapp* for example, we send its code as a POST request to the URI */install?myapp*. The install-resource then stores the code on disk and adds a new subresource */install/myapp* that represents the new app. From this app, we now can create one or more instances.

To create a new instance of an app, send a POST request to it that contains at least the definition for the name property in the form `name = myname`. The name must be different from all other existing instances of any app. It is also possible to define further properties in the form `key = value`. In particular one might want to define any predefined instance-properties or start parameter for the app-instance. If a predefined property is not explicitly defined, a default value will be chosen. Actinium then creates two new resources. First, a new subresource for *app/appconfigs* that holds the instance-properties. Second, a new subresource for *apps/running* that contains the app itself.

To remove a single app-instance, send a DELETE request to the corresponding configuration resource. To update the instance-properties of a single app-instance, send a POST request to the corresponding configuration resource with the new property values in the form `key = value`. To remove an app, send a DELETE request to it. Actinium then will remove the app, delete it from disk, and also stop and remove all instances of that app. To update an app, send a PUT request with the updated program code to it. Actinium then will update the app and restart all its instances.

2.4 Isolation of apps from the underlying system

For a developer, JavaScript is a powerful tool for creating apps. Developer can come up with anything and realize it as an app. Freedom for a developer bears, however, a tricky challenge for

a) Grouped by resource type.

b) Grouped configs layout.

c) Grouped apps layout.

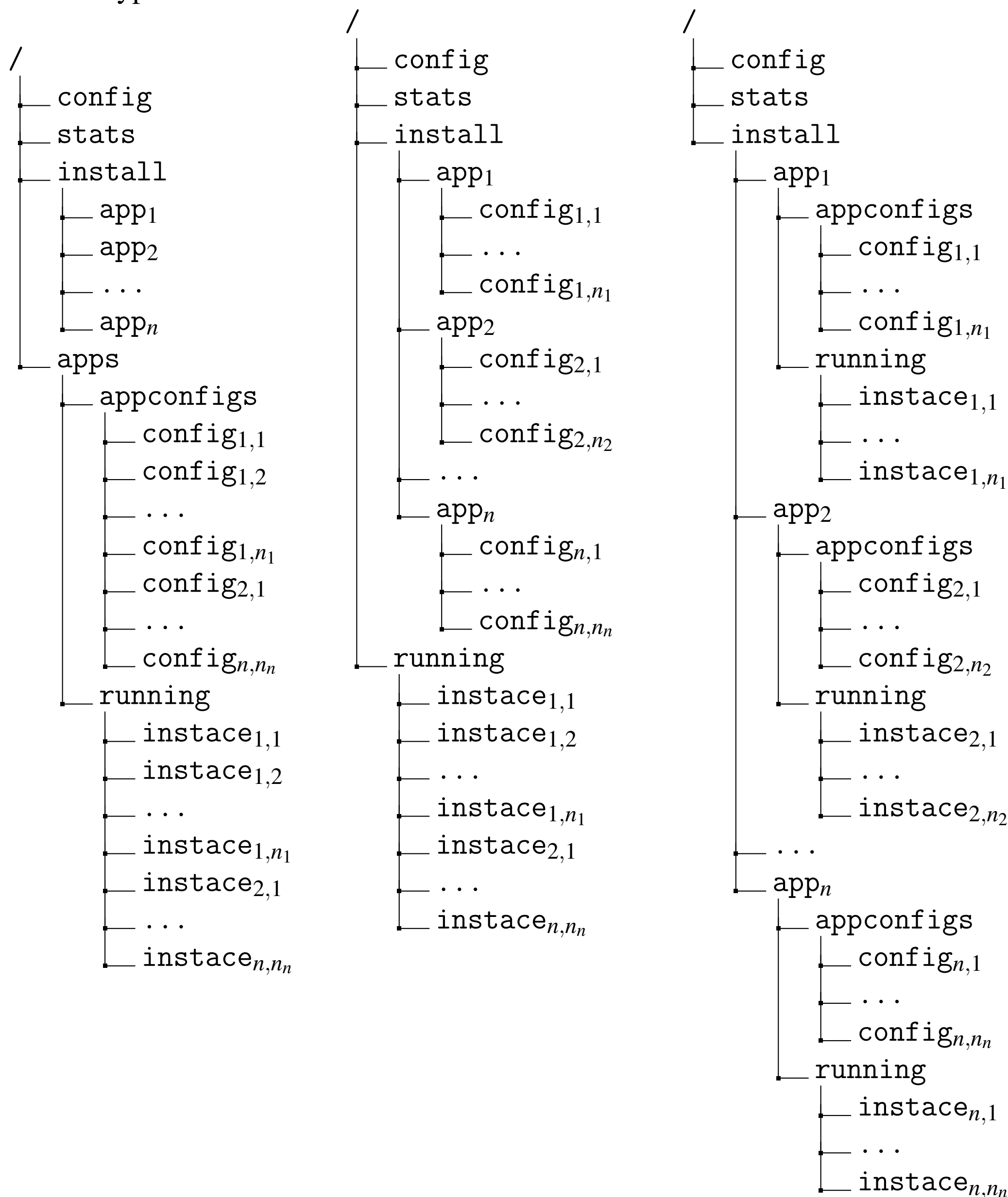


Figure 2.1: Resource layouts: a) Group together the resources for configurations, running apps and installed apps respectively. b) One group per app and its instances' configurations and one group for all running app-instances. c) One group per app and its instances' configurations and running app-instances.

Actinium. It has to expect anything imaginable in JavaScript: Malfunctions, exceptions, endless loops, deadlocks, and even evil behavior. For this project we omit evil intention of a developer and assume malfunctions happen due to a careless but unintended erroneous implementation. For Actinium to be able to handle such kinds of programs, apps must be isolated from the underlying

system, in particular from Californium, and from other apps. If one app fails, no other app or component of the system must be affected by the failure.

Californium is the app's mailman. Every request targeting an app passes through it. The core work of Californium, handling requests, is done by the so called ReceiverThread. When a request arrives, Californium looks for the destination resource whereon the ReceiverThread executes this resource's request handling method. As long as the ReceiverThread executes the request handler and doesn't yet return, it is not available for other incoming or outgoing requests. If the request handler takes a very long time to finish or is even stuck in a deadlock or busy-loop, the ReceiverThread is no longer available. Without ReceiverThread, Californium stands still. Therefore we essentially have to protect the ReceiverThread from malfunctioning apps.

Actinium executes all app-instances concurrently, i.e., in a separate thread, thus maximizing parallelism among them. An app-instance's thread first executes the app's JavaScript code and initializes possible variables and dependencies to other apps. Furthermore, every app-instance holds a queue for requests. If a request for an app-instance arrives, the ReceiverThread puts it into the app-instance's request queue. It is then the app-instance's executing thread that executes the corresponding request handler, see Figure 2.2.

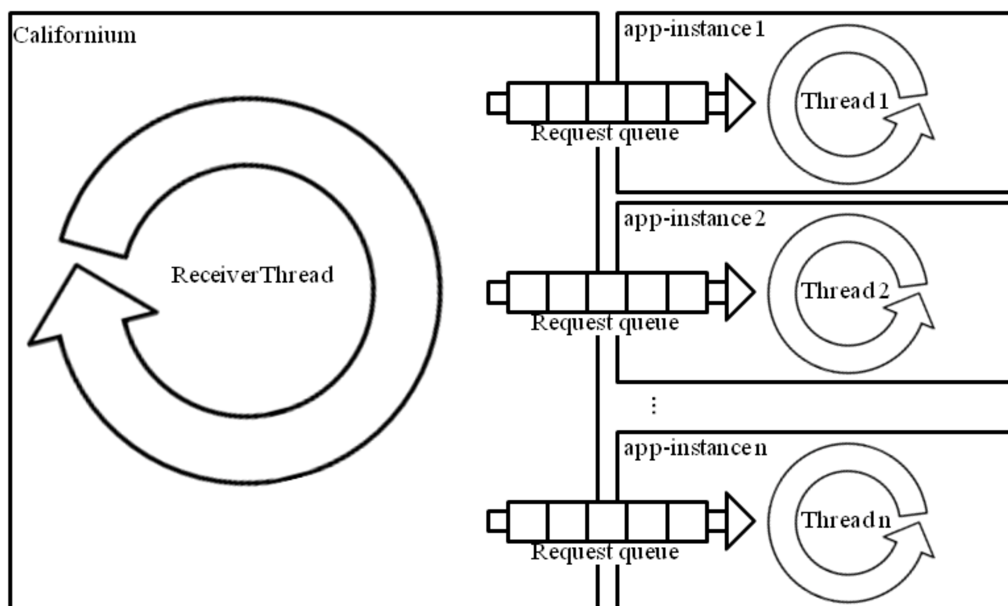


Figure 2.2: Californium's ReceiverThread receives requests and passes them through a queue to the target app-instance. The app-instance's thread then executes the app-instance's request handler.

This design maximizes the ReceiverThread's responsiveness and protection from failures within app-instances. If an app-instance crashes, it no longer responds to requests but doesn't affect any other component of Actinium. One might consider limiting the request queue to some number of requests and, if full, drop further requests. This would prevent Actinium from unboundedly aggregating requests after an app-instance has crashed.

We also considered using two threads per app-instance. One for executing the app's JavaScript code which then would be allowed to run infinitely and another one for executing the app-instances request handlers. This led to two problems, however. First, a request handler might be executed before the corresponding variables have been initialized. Second, the developer would have faced an inherently multi-threaded environment with all associated challenges. Therefore we discarded this idea.

3 Implementation

In this section we omit the distinction between an app and its app-instances. Thus, the term app is used interchangeably with the term app-instance.

3.1 CoAPRequest

Actinium's communication bases on the CoAP protocol. If an app wants to send a CoAP request to another resource, Californium provides classes and methods to do so. For our apps, written in JavaScript, however, we strive for an even simpler, yet powerful, API. We introduce the CoAPRequest object, an API similar to the well-known XMLHttpRequest API of AJAX [11]. CoAPRequest provides the same main methods `open(...)`, `send(...)`, `abort(...)` with equivalent behavior as XMLHttpRequest. Furthermore it is possible to get and set request headers. After certain events, homonymous listeners as in XMLHttpRequest such as onreadystatechange, onload, onerror and ontimeout are called if assigned to by the developer. In contrast to XMLHttpRequest, the states HEADERS_RECEIVED (numeric value 2) and LOADING (numeric value 3) have been omitted, since CoAP is a UDP-based protocol where the content arrives atomically. The CoAPRequest's field status corresponds to the CoAP status (e.g. "Content" with value 2.05), while the field *httpstatus* corresponds to the according status from HTTP (e.g., "OK" with value 200) [10]. Following is a simple code demonstration, how to fetch data from a CoAP resource.

```
var client = new CoAPRequest();
client.onreadystatechange = handler
client.open("GET", "coap://mydomain/myresource");
client.send();

function handler() {
  if (this.readyState==this.DONE) {
    if (this.status==this.Content) {
      processData(this.responseText);
    }
  }
}
```

The demonstration example uses an asynchronous request. If the app receives a response, it calls the `onreadystatechange` listener with readystate `DONE` (numerical value 4) and the listener `onload`.

CoAPRequest also provides synchronous requests. In that case, the executing thread stops until a response is available. It is also possible to set a timeout for the CoAP request. If a timeout occurs the app calls the listener ontimeout if defined, and sets an error flag, see appendix A.

3.2 Encapsulated commands

Actinium enriches the JavaScript environment with a few auxiliary functions to enhance developers' productivity. In the JavaScript code, those functions are accessed via a predefined object, called app. For instance, to get the time in milliseconds or nanoseconds since 1970, we use the command app.getTime() or app.getNanoTime() respectively.

app.dump and app.error: For debugging the app or logging some events, the functions `app.dump(...)` and `app.error(...)` give access to the standard output and error stream respectively.

app.getProperty: The function `app.getProperty(key)` returns the value of the given property from the instance-properties of the app or `null` if not found. The function `app.getProperty(key, default)` works likewise but returns the default value if no property is found. Note, that we restrict apps to only read from those properties but not write to them.

app.root: The object `app.root` serves as root of the resource hierarchy. It conforms to the API for resources from Californium. With app.root.addSubresource(...), an app adds its own subresources as children to the root.

onget, onpost, onput, ondelete: For an app to bind a function as GET-, POST-, PUT- or DELETE-request handler to a resource, we use the function-objects `onget`, `onpost`, `onput` and `ondelete` respectively. For instance, a simple Hello world program is realized by only few lines.

```
app.root.onget = function(request) {
  request.respond(this.Content, "hello world");
}
```

onunload: In some cases, when an app is shutting down, it might want to do some cleanup before. For instance, an app should cancel observation relationships to other resources so that other resources no longer send messages to the app. The object app provides the function `onunload` to which any function can be assigned to and which Actinium will call when the app shuts down.

Auxiliary Functions	
Function	**Description**
`root`	The CoAP resource that represents this app.
`dump(String)`	Prints to the standard output stream.
`error(String)`	Prints to the standard error output stream.
`shutdown()`	Stops the app.
`restart()`	Restarts the app.
`sleep(Integer)`	Sleeps for the specified amount of milliseconds.
`getTime()`	Returns the current time in milliseconds (since 1970).
`getNanoTime()`	Returns the current value of the most precise available system timer, in nanoseconds.
`getProperty(Key)`	Searches for the property with the specified key in this property list. The method returns null if the property is not found.
`getProperty(Key, Default)`	Searches for the property with the specified key in this property list. The method returns the default value argument if the property is not found.
`onunload`	Will be called, when the app shuts down.

Table 3.1: Auxiliary Functions for JavaScript apps

3.3 The stats resource

The stats-resource provides information about the apps which can be retrieved by a GET request. It uses Java's ThreadMXBean [1] to measure the time the CPU has spent within the app. Whether ThreadXMBean is supported and the accuracy of the results depends on the JVM and the operating system. We have modified Californium's request delivery mechanism to record all incoming requests. The stats-resource counts the requests and its payloads in bytes for each app and for each its subresources. This information gives us a fine grasp, about an app's CPU usage and the traffic towards the app. If Actinium stresses the CPU, an end-user could check which app causes the high CPU load. Furthermore he could check, whether the app is congested by many incoming requests and stop the app if needed.

[1] http://docs.oracle.com/javase/6/docs/api/java/lang/management/ThreadMXBean.html, December 12, 2011

4 Performance Evaluation

Actinium is designed to host and execute application logic and we require an app reach a satisfying performance. In this section, we evaluate the performance of Rhino for JavaScript apps in Actinium. First, we compare JavaScript apps running on Actinium in Rhino to JavaScript apps running on Firefox. Second, we compare JavaScript apps to apps, written in Java. We run our tests on a HP EliteBook 8530w with Intel(R) Core(TM)2 Duo CPU T9400, 2.53 GHz and 4 GB memory with Windows 7 Professional x64 Service Pack 1.

4.1 Benchmark Methodology

We compare three different run-time systems:

- Java 1.6.0_21 with Java HotSpot 64-Bit [1]
- Actinium with Rhino 1.7R3 [2]
- Firefox 10.0.1 with Jägermonkey [3]

We have compiled the java programs with javac -g:none -O to exclude debugging information and turn on optimization, and ran them with java -hotspot to activate the just-in-time compiler within the JVM. Rhino 1.7R3 from June 2011 is currently the most recent version and supports JavaScript 1.8. Mozilla Firefox 10.0.0.1 uses the JavaScript engine SpiderMonkey with the method-JIT JägerMonkey [4], written in C/C++ [5].

Since JavaScript apps are supposed to be light-weight, we don't expect heavy I/O operations for instance but rather arithmetic computations. We use three different algorithms in our benchmark:

- Recursive Fibonacci algorithm [6]
- Quicksort
- Newton's Square Root algorithm [7]

We write all three algorithms equivalently in JavaScript and Java.

[1] http://www.oracle.com/technetwork/java/javase/6u21-156341.html, February 12, 2012
[2] http://www.mozilla.org/rhino/, February 10, 2012
[3] https://wiki.mozilla.org/JaegerMonkey, February 12, 2012
[4] http://blog.mozilla.com/dmandelin/2010/09/08/presenting-jagermonkey/, February 12, 2012
[5] https://developer.mozilla.org/en/ SpiderMonkey, February 12, 2012
[6] https://wiki.mozilla.org/JaegerMonkey, February 12, 2012
[7] http://en.wikipedia.org/wiki/Newton%27s_method#Square_root_of_a_number, February 12, 2012

The Recursive Fibonacci algorithm computes the Fibonacci number by repeatedly, recursively calling itself. This causes a huge number of function calls for which we want to see, how efficient the run-time systems perform. We measure the algorithm's time for inputs 30, 35 and 40. Quicksort sorts an array of double-precision floating point numbers. This benchmark focuses on how efficient the run-time systems handle memory accesses. We measure the algorithm's times to sort 10^4, 10^5, 10^6 and 10^7 numbers. Newton's Square Root algorithm is a fixed-point algorithm that computes the square root for a number. Since the result doesn't matter, we always use just eight iterations. With this algorithm we compare how efficiently the run-time systems deal with floating point operations. For an input n, we compute the square roots of all natural numbers from 1 to n. We measure the algorithm's time for inputs 10^5, 10^6 and 10^7.

We use the K-best measurements method [8] with K=10. Thus, we run every test for each input on every run-time system 10 times and take the minimal elapsed.

4.2 Results

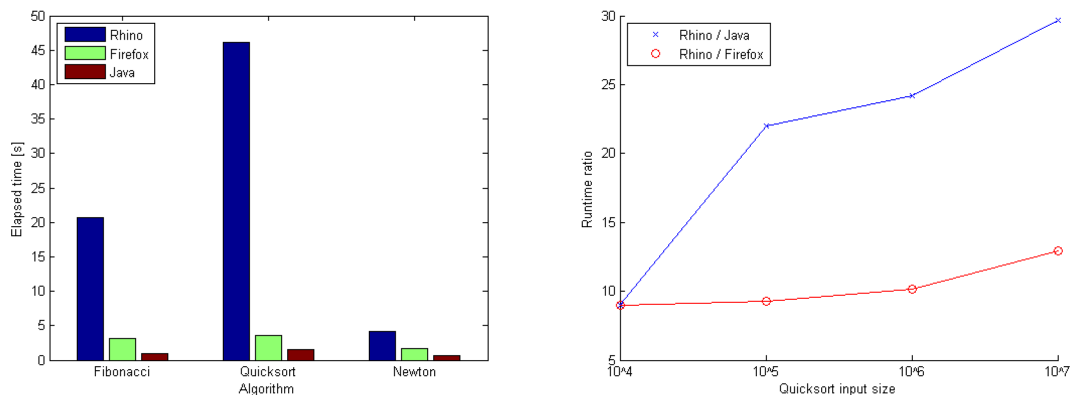
Fibonacci					
Input	Rhino	Firefox	Java	Rhino/Firefox	Rhino/Java
30	162 ms	25 ms	9 ms	6	18
35	1796 ms	278 ms	87 ms	6	21
40	20738 ms	3105 ms	969 ms	7	21

Quicksort					
Input	Rhino	Firefox	Java	Rhino/Firefox	Rhino/Java
10^4	18 ms	2 ms	2 ms	9	9
10^5	242 ms	26 ms	11 ms	9	22
10^6	3165 ms	312 ms	131 ms	10	24
10^7	46058 ms	3560 ms	1552 ms	13	30

Newton					
Input	Rhino	Firefox	Java	Rhino/Firefox	Rhino/Java
10^5	42 ms	16 ms	7 ms	3	6
10^6	423 ms	162 ms	67 ms	3	6
10^7	4219 ms	1625 ms	671 ms	3	6

Table 4.1: Elapsed times for Fibonacci, Quicksort and Newton. For each input, every algorithm runs in the run-time systems Rhino, Firefox and Java (left). The ratio between the times of Rhino and Firefox and the ratio between the times of Rhino and Java (right).

[8] http://www.cs.rochester.edu/ cding/Teaching/252Spring2003/LectureNotes/Time.ppt



a) Algorithms with highest input values. b) Runtime ratio between Rhino and Firefox and Java for Quicksort.

Figure 4.1: a) Runtimes for all engines for all algorithms with the highest tested input values (40 for Fibonacci, 10^7 for Quicksort and Newton). Rhino clearly is slower than Firefox and Java but it also depends on the algorithm. b) The ratio between the runtimes of Rhino and Firefox and Java for Quicksort. The ratio increses for higher input values. Firefox and Java scale better than Rhino.

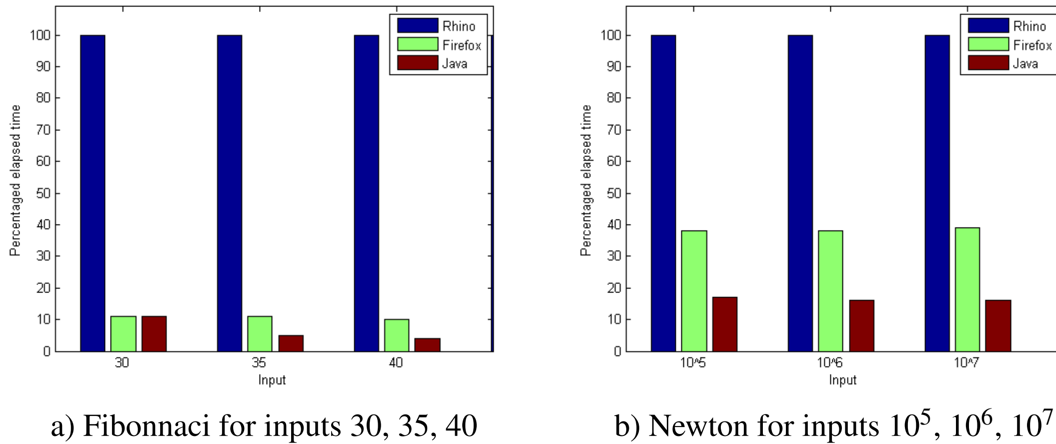
Table 4.1 clearly shows that the elapsed time for Rhino to execute an algorithm is higher than the elapsed times of Firefox and Java. Figure 4.1 a) shows the performance differences of the engines for the elapsed times for the highest inputs per algorithm. Clearly, Rhino takes a multiple of the time of the other two engines.

In contrast to Fibonacci and Newton, Quicksort's performance relative to other engines also depends on the array size. The ratio between Rhino's and the other engine's elapsed times grows for bigger arrays. For smaller arrays, Rhino takes up to 9 times longer than Firefox and Java. For a huge array of 10^7 double-precision numbers Rhino becomes 13 times slower than Firefox and even 30 times slower than Java as shown in figure 4.1 b). The memory management of Rhino is not as efficient as that of the others. However, it is very unlikely an app ever has to sort such big arrays. On our test computer we easily can sort 10^5 doubles within a second with a non-optimized Quicksort algorithm. This should suffice by far for any reasonable app.

The relative performance difference depends severely on the algorithm. Rhino roughly takes 6 times longer than Firefox and 20 times longer than Java to perform Fibonacci as shown in see figure 4.2 a). Rhino's mechanism for invoking methods clearly is not as fast as Firefox's JavaScript engine and the Java runtime engine.

Rhino performs better for the Newton algorithm. Rhino's performance is only 3 times slower than Firefox's and 6 times slower than Java's performance as shown in figure 4.2 b). Unit operations like plus, minus, multiplication and division can be performed equally fast for any run-time. The performance differences merely arise from loading and storing values in variables.

A real-world app will not consist of only one kind of algorithm as the one-sided algorithms above but rahter a mix of it. Therefore, performances of real-world apps are expected to vary



a) Fibonnaci for inputs 30, 35, 40

b) Newton for inputs 10^5, 10^6, 10^7

Figure 4.2: a) Percentaged elapsed times of each engine for Fibonnaci compared to Rhino as 100%. b) Percentaged elapsed times of each engine for Newton's Square Root algorithm compared to Rhino as 100%.

between lower and upper measurement boundaries of these algorithms. For a client that has sent a request to an app not only the time to handle the request matters but also the transmission time for the request and the response. To compute the complete waiting time of a client for a response, we have to add the processing time of the app to the latency of the network. Kovatsch [5] shows that this latency strongly depends on the size of the payload and the amount of hops a message has to pass. Kovatsch's measurements vary between about 50 ms latency for 64 bytes payload and direct network connection between client and server and about 1600 ms latency for 512 bytes payload and a network connection over 3 intermediate hops between client and server. No matter how fast a request is handled, the network latency prevails as lower bound for the response time.

Rhino uses Java reflection to invoke JavaScript functions and is therefore not as efficient as Firefox' JavaScript engine, written in C/C++. Apps that require very high throughput or even real-time would not be implemented in JavaScript anyway but with Java or even C. It is clear that apps, written in JavaScript, cannot achieve the performance of compiler-optimized Java code. However, Actinium executes each app in a separate thread. Californium's ReceiverThread delivers requests sequentially, but the handling of requests for an app is done by its thread. Hence, apps inherently handle incoming requests concurrently and thus can utilize multicore systems. A single app written in Java may be much faster than a JavaScript app, but as multicore systems evolve and a host for Actinium includes more and more CPUs in the future, JavaScript apps catch up. In contrast, programming Java resources that handle requests concurrently in separate threads requires a major effort from a developer, cooperation among all resources and yet is error-prone.

When deciding for a programming language, a tradeoff between performance and convenience for developers is required. In an environment of simple apps for which even a lower performance is enough by far, convenience must be the dominant factor for the design of a programming language. JavaScript serves this purpose best, as it is a simple yet powerful and very dynamic high-level language. Developer conveniently can design robust, reliable, traceable, portable and legible apps and maintain them at runtime.

4.3 Outlook

One of the major challenges for Rhino is its support for dynamic typing. “When developers write engines for dynamically typed languages that run in the JVM, they have to satisfy the requirements of the Java bytecode that the JVM executes.” [9] Java 7 introduces a new bytecode instruction invokedynamic and a new method linkage mechanism. By using this new instruction, the developers of Rhino expect a significant speedup that makes Rhino much more competitive [10]. If Rhino implements this optimization in a future version, its performance for our three algorithms has to be reevaluated and will most likely perform much better.

[9] http://java.sun.com/developer/technicalArticles/DynTypeLang/, February 18, 2012
[10] http://www.infoq.com/news/2011/07/rhino-invokedynamic, February 18, 2012

5 Conclusion

Actinium, the implementation of our app-server design, builds upon the Californium (Cf) CoAP framework and extends it with powerful features. Lightweight apps can be written conveniently in JavaScript and dynamically installed, updated, removed, and monitored. Configurations allow the definition of parameters and boundaries for apps and provide a strong control mechanism. Actinium produces a smart resource layout that structures all components and updates it automatically according to installation changes. The installation process is straight forward and allows reusing an app's code multiple times, thus saving system resources and enhancing maintainability.

Actinium keeps apps in a sandbox and serves as a layer between them and the outside world. It concurrently executes each app in a separate thread and passes all incoming requests to them through a queue, thus protecting Californium and apps from malfunctioning apps. Actinium arms JavaScript apps with a convenient though powerful API, thus maximizing efficient development of robust apps. The CoAPRequest API is kept similar to the well-known XMLHttpRequest API and guarantees reliable and traceable communication with other CoAP resources.

Actinium's integrated JavaScript engine Rhino is clearly slower then Firefox's engine or the JVM for Java apps. This depends strongly on the algorithm, however, and is supposed to improve drastically due to better scripting support of Java 7. Moreover, since Actinium executes apps concurrently, they can utilize multicore processors. For reasonable apps, Rhino's performance suffices by far.

Actinium is extensible. Further engines for apps written in other languages can be integrated, using Actiniums interfaces. Configurations are very flexible and adaptable to future changes and extensions. The resource layout is elaborate and provides numerous nodes for extensions.

Acknowledgements

I would like to thank Matthias Kovatsch for his dedicated support. I enjoyed our talks about the principles and philosophy behind Californium, CoAP and the Web of Things in general.

A JavaScript and CoAPRequest Examples

Actinium provides a rich API to JavaScript apps. We show a few example apps to give the reader a rough grasp how to write a JavaScript app for Actinium.

Hello World App

The app responds to a GET request with "Hello World".

```
app.root.onget = function(request) {
  request.respond(this.Content, "Hello World");
}
```

Handling requests

The app counts POST requests. If a POST request arrives, the handler accepts the request, extracts the payload from the request, increases the counter by one, computes the result and sends the payload back together with the counter.

```
var counter = 0;

app.root.onpost = function(request) {
  // accept request
  request.accept();

  // get payload and increase counter
  var payload = request.getPayloadString();
  counter++;

  // compute result
  app.sleep(1000);
```

```
  // respond
  request.respond(this.Content, "payload: "+payload+", counter = "+counter);
}
```

Subresources

Create a new subresource, add it to app.root and define a handler for GET requests.

```
res = new JavaScriptResource("subresource");
app.root.addSubResource(res);
res.onget = function(request) { ... }
```

Send a GET request to a CoAP resource

Fetch data from a CoAP resource over the network using the CoAPRequest API.

```
var client = new CoAPRequest();
client.onreadystatechange = handler;
client.open("GET", "coap://mydomain.myresource");
client.send();

function handler() {
  if (client.readyState==this.DONE) {
    if (client.status==client.Content) {
      processData(client.responseText);
    }
  }
}
```

Send a POST request to a CoAP resource

Send a POST message to a CoAP resource.

```
function sendMessage(message) {
  var client = new CoAPRequest();
  client.open("POST", "coap://mydomain.myresource");
  client.setRequestHeader("Content-Type", "text/plain");
  client.send(message);
}
```

CoAPRequest Timeout

Send a POST request using a timeout and react to network problems and successful transmissions.

```
var client = new CoAPRequest();
client.open("POST", "coap://mydomain.myresource");
client.timeout = 1000; // one second

client.onload = successhandler; // called if we successfully get response in time
client.onerror = errorhandler; // called if there is a network error
client.ontimeout = timeouthandler; // called if no response in time

client.send(data);
```

Bibliography

[1] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and V. Bahl. The home needs an operating system (and an app store). In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 18:1–18:6, New York, NY, USA, 2010. ACM.

[2] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. volume 2, pages 115–150, New York, NY, USA, May 2002. ACM.

[3] M. Kovatsch. Demo abstract: Human-coap interaction with copper. In *Proceedings of the 7th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2011), Barcelona, Spain*, June 2011.

[4] M. Kovatsch. Firm firmware and apps for the internet of things. In *Proceedings of the 2nd ICSE Workshop on Software Engineering for Sensor Network Applications (SESENA 2011), Honolulu, HI, USA*, pages 61–62, New York, NY, USA, May 2011. ACM.

[5] M. Kovatsch, S. Duquennoy, and A. Dunkels. A low-power coap for contiki. In *Proceedings of the 8th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS 2011), Valencia, Spain*, pages 855–860, Oct. 2011.

[6] M. Kovatsch, M. Weiss, and D. Guinard. Embedding internet technology for home automation. In *Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2010), Bilbao, Spain*, Sept. 2010.

[7] K. Kuladinithi, O. Bergmann, T. Pötsch, M. Becker, and C. Görg. Implementation of CoAP and its Application in Transport Logistics. In *Proceedings of Extending the Internet to Low power and Lossy Networks (IP+SN 2011)*, Chicago, USA, April 2011.

[8] M. Lanter and M. Kovatsch. CoAPRequest API. Institute for Pervasive Computing, Department of Computer Science, ETH Zurich, 2012.

[9] D. Pauli and D. Im Obersteg. Californium. Institute for Pervasive Computing, Department of Computer Science, ETH Zurich, 2011.

[10] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). In *draft-ietf-core-coap-08*, 2011.

[11] A. van Kesteren. XMLHttpRequest Level 2. January 2012.