Towards Reactive Information Systems and their Services

Master Thesis

Author:
Dominic Bosch

Supervisors:
Prof. Dr. Helmar Burkhart
Dr. Martin Guggisberg

June 12, 2014

UNI
BASEL

Abstract

The Web is a heterogenous collection of data and services. A large set of this Web information space is dynamic and changes within it are covered by the event space. In our work we describe a conceptual model, which makes the Web reactive by listening for events and executing actions in the information space, according to predefined rules. Since a large part of the event space is not ready to be actively distributed to listeners,

Acknowledgment

I would like to express my deep gratitude to my master thesis advisor, Prof. Dr. Helmar Burkhart, for his patience and constant encouragement to keep going further. I also want to thank Dr. Martin Guggisberg for his ever helpful advices, hints to existing technologies and guidance through the whole process of this thesis. Besides my advisors, I would like to thank the members of the research group: Danilo Guerrera, Antonio Maffia, Alexander Gröflin and Robert Frank for their help, many interesting discussions and inputs. My thanks also go to Manus Bonner for reading through this and coming up with well formed sentences. Finally I would like to thank my family; my parents Silvia and Peter Bosch, my sister Svenja, her husband Andreas Buser, their seven months old son Nico and my girlfriend Kathrina for their inspiring ideas, the joyfulness, irresistible smiles and financial support they gave me on my journey.

Contents

Chapter 1

Introduction

The Web is an ever growing institution, in all aspects that it covers. Research on behalf of the Web attracts a great deal of attention, even more since modern life is already impossible without it. The number of information and functionality providers for the Web is growing in the whole spectrum from bigger computing centers down to smaller devices. Computing centers, growing in size and quantity, allow for massive amounts of data to be stored and accessed, but they also enable the construction and accessibility of more complex functionalities. Ever smaller devices provide informations and functionalities to the Web, through their quickly increasing number. Many of them are even providing access to the Web itself and thus leverage the effect of the growing Web. For example mobile phones can act as a hotspot to grant Web access to others through WiFi. All the smart Things with access to the Web start to form the Web of Things. This can be everyThing from a temperature sensor to all electronic devices within a house. These Things do not only provide sensor data but they can also be easily controlled through the Web.

Confronted with this rapid growth of the Web, an increasing number of human beings is exposed to it in their daily life and they get literally flooded with data and possibilities to govern them. Even though they have access to all these services and data in the Web, they often lack the knowledge, necessary time or right approach to weave them together. Users need ways to get appropriate informations, in the right moment, automatically and in a condensed matter that suits them best. They have to be able to automate tedious tasks, e.g. detecting relevant changes in information resources and react on behalf of such changes. *(More concrete example?)* This requires the identification of and filtering for user-specific information, appropriate timing, assembly and preprocessing of different information resources and finally the forwarding of outcomes to services in the web.

Users don't want to be bound to specific services or applications, they want to use their preferred one, which they are used to and which helps them best to fulfill their needs. Some of these services in the web offer ways to spread their data to other predefined services, but in very limited amount and parametrization. Therefore users end up mashing informations or functionalities from different locations within the Web by hand. This often means to execute similar tasks repeatedly by hand. Moreover the completion of such a task suffers from latency due to the deferred detection of the initialization or because the user is just not able to execute the work at that time. And also if the bits and pieces that form such a task could be separated, it is likely that some parts could be reused for other tasks or even by other people to get similar work done.

Since the data and the functionality already exists in the Web, the users could automate their work to some extent by orchestrating services and data. Even though the access to services and data gets simpler, the average user is not able to fully exploit the Web. Another challenge is that often a lot of effort needs to be invested, in understanding how the service works, before it can be fully exploited. It is desirable to retain manageable mashing of services and data, while still exploiting their full potential. There is a lot of research that goes towards an easy to orchestrate the Web, but they are either often complicated to wield themselves, mere data copy or static service compositions.

A big part of the informations, that become available to the users, are short-lived informations corresponding to certain state changes, and can therefore be modeled as events. In this thesis we to introduce an event-driven conceptual model that enables the programmability of the Web and thus imposes reactivity on the Web. We claim the whole Web as our information space, in which we listen for triggered events and in which we execute actions as a result. Such an user-specific reactivity allows a personalization of the whole Web and a tool to govern its information flood. This would allow users to orchestrate the Web in an intuitive way and to tailor reactivity to their needs. Current orchestration approaches concentrate on data flow rather than on event flow, which are mere data copy/paste tasks than reactivity. This makes us believe that an event-based conceptual model can overcome certain shortcomings of the existing solutions.

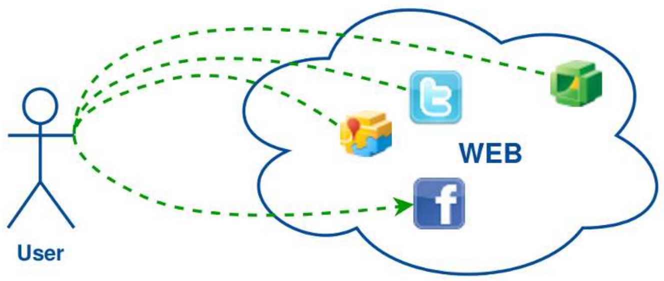

Figure 1.1: Users orchestrate the Web's Data and Functionality

Chapter 2

Related Work

In this chapter we will give a brief overview over resources offered through the Web, which are of interest to us. We then introduce the Event-Condition-Action(ECA) paradigm, an event-driven approach to impose reactivity. And finally we point out existing rule languages and engines that exploit the ECA paradigm.

2.1 Data and Functionality Resources on the Web

The term service in the context of the Web is ambiguous and there have been a lot of different approaches to offer services within the Web, some of the latest used in cloud computing being Platform as a service (PaaS), Software as a service (SaaS) and Infrastructure as a service (IaaS). The term Web Service is widely understood as interfaces for communication between applications over a network, but has seen many specific adoptions. We will point out some main research areas on service-orientation within the Web and give an idea of what kind of services in the Web we intend to orchestrate with our conceptual model.

Remote execution of programs on other computers has always been a strong research area. And with the coinage of the term World Wide Web [5], there were also trends towards services offered through the Web; computers waiting in the Web for a request in order to execute some application logic and return an answer. The encapsulation of functionality into services [28] in order to offer them to other applications is called service-oriented architecture (SOA) [29]. Adopting SOA internally to an application means splitting the application into smaller pieces, which then communicate via these services among each other. This does not only provide robustness, it also allows the reuse of functionality through services. Moreover these services can be offered to other applications and also to the Web, thus allowing others to access certain functionality or even the whole application. All nodes in the Web are stand-alone entities, which offer services of some sort, be it a webpage, data, instant measurements or functionality. This makes the Web itself a service-oriented architecture and all these services are naturally services in the Web. It is for its advantages that SOA has received a great deal of attention and has been widely adopted, mostly throughout the Web. This lead to an increasing number of Web accessible services and their compositions, the so called Mashups. An empirical study [22] on a directory, which they call the *"[...] most active Web APIs and mashups collection"*, and data taken from this resource (depicted in Figure 2.1) seem to underline a growing popularity, at least in terms of publishing services through this directory.

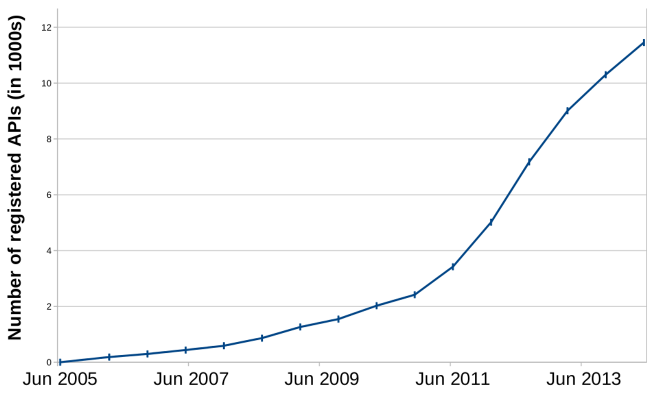

Figure 2.1: Number of registered APIs in the ProgrammableWeb directory by date

2.1.1 Requesting Services in the Web

An early adoption of the service concept to computers were the Remote Procedure Calls (RPC) [7]. Through RPC a piece of code can be executed on a different machine, than the one which is calling the procedure. It is basically achieved via inter-process communication and doesn't necessarily require the Web. Even more since when RPC was invented, the World Wide Web wasn't even postulated by Tim Berners-Lee. RPC also found its use in grid computing [36] and through this, opened doors into the field of distributed computation. The RPC paradigm isn't bound to certain technologies and thus, has been implemented in a lot of different programming languages. These implementations were tightly bound to the respective language that was used, which resulted in incompatibility among them. It became necessary to enhance RPC's in order to get cross platform compatibility. The abstraction of RPC through the Extensible Markup Language (XML) [10], which is called XML-RPC, compatibility between services that used different technologies was easier to achieve.

Since XML-RPC was held relatively simple but received a lot of attention, it was further enhanced. Together with additional proposed functionality, XML-RPC heavily influenced Simple Object Access Protocol (SOAP) [9]. SOAP is accompanied by the Web Service Description Language (WSDL) [14] which is used to describe the interfaces to SOAP services. Through SOAP and WSDL a client for the service can issue a request for the WSDL information of the service and retrieves all interface specifications he requires in order to issue a call to the actual service. The service specifications are then incorporated into the existing application as if it is a local function call. SOAP has found its applicability in business applications [4] and was enhanced with a lot of industrial standards, also called the "WS-*" specifications, e.g. WS-Addressing, WS-Policy or WS-Security.

Another initiative that aimed for eased communication between different platform is the Common Object Request Broker Architecture (CORBA) [20]. As the name already suggest it is an object-oriented approach and it allows the exchange of whole objects. CORBA relies on its communication layer, the Object Request Broker (ORB), which forms the basis of its architecture. The platform-specific ORB's provide the communication abstraction, which free the application from platform dependencies. Similar to SOAP's WSDL, CORBA has its Interface Definition Language (IDL) to provide information about the objects to be offered and accessed. An object is instantiated by an application and the interface to this instance is offered through the ORB. Another application attached to the ORB can then access all public variables, data structures and functions of this object. This means not only remote access to variables and data structures, but also remote function invocation. CORBA requires the implementation of object-oriented mechanisms in programming languages which aren't object-oriented. This can be technically difficult and become an eventually tedious task. CORBA allows communication between applications written in different programming languages and which are running on the same physical computer, as well as the communication between different computers in the same network. With the Internet Inter-ORB Protocol (IIOP) it is also possible to connect ORB's over the Web. Through this, the offered objects can become services in the Web, but they are shielded by the ORB.

2.1.2 Services in the Web become Web Resources

All the afore mentioned approaches require a specific protocol and are therefore incompatible with each other. For this reason and its simplicity, an architectural style has gained popularity which frees application from this constraint: Representational State Transfer (REST) [17]. REST concentrates on the roles of components and on constraints upon interactions between them. An important architectural constraint is that all communication is stateless, which means for a client-server communication, no state is stored on the server. Therefore all informations required for a single interaction need to be provided within one request. This allows for the definition of simple and well-defined interfaces, since responses are not bound to a certain session state. Services within the Web that adhere to the REST architecture are called RESTful Web services. RESTful Web services provide access to their data and functionality through grouped Web Resources, which can be identified vie Uniform Resource Identifiers (URI). Simple access to services in the Web without communication overhead and required negotiation before using it, increased REST's popularity and spread it into more application fields. There is for example the upcoming concept of the Web of Things [21], which aims to incorporate smart things (e.g. tagged things, sensor measurements, device controllers, etc.) into the Web through REST interfaces. REST brings advantages into the context of smart things connected to the Web, because incompatible standards and protocols were used by different manufacturers of such things.

2.1.3 Composing Services in the Web

Webpages emerged into dynamic sites on the web through the upcoming of scripting languages to control the browser and the webpage itself. With all their infrastructure in the background on the server they became literally applications. These Web applications (Web Apps) went even more responsive with the advent of asynchronous calls from the browser to the server, which allows to load data into the current webpage while the user is interacting with it. Those asynchronous calls are requests to services, which act as the application programming interface API to the Web App (Web APIs) which sits on the server. As a side-note, the term Web API not only comprises server-side interfaces but also client-sided ones (e.g. the browser), after all they are also interfaces to the Web. For server-side Web APIs this means that these services can be accessed from other entities in the Web than just browsers, which eases application to application communication. Basically a Web App can be controlled without the user interface, which is often delivered by the provider of the application. Imagine not going to the Google webpage anymore to make a search and crawling through the results, but you have your own application doing it for you and processing the results instantly. There is a trend of Web App providers to publish their Web API in order to grant easy access to it. This lead to an increase of the number of Web App Mashups in the past few years.

Mashups combine data and functionality of more than one service in the Web in a new site. Simple services from different sources can be combined into more powerful ones, which can in turn again be composed and so on. These service compositions assemble data and services in a novel way which provides a new perspective. Ever since services were accessible in a more or less convenient way, Mashups have been developed as well. On of the first Web service Mashups [32], was invented in the same year after Google Maps came up in 2005. It was a webpage that displayed CraigsList's rental houses on a Google Map. At that time no Web API was available that provided easy access to these two services, but there was an advantage to be seen from everybody being able to create a Mashup through publicly available services. Such Mashups are often a read-only fixed wiring of different services that provide a new view on specific data. Some recent Mashup examples, taken from the ProgrammableWeb [31] collection, are:

- Wifi and Plugs: MapBox, Google Docs and Import.io API's used to display where Wi-Fi and plugs are available in London.
- MapLight: GovTrack.us and OpenSecrets API's used to combine political results with financial contributions to show how capital contributions to influence politics affect voting.
- Shared Count: Facebook, LinkedIn, Pinterest and Twitter API's used to display informations about how well spread a URL is on social media sites.

But also a number of studies [13][23][35][37] made efforts towards personalized Mashups, where users are capable of choosing what and how to link in order to enhance Web resources according to their needs. These flexible Mashup applications often provide methods to access user-specific functionality within remote Web Apps, which makes them even more user-centered and customizable.

2.1.4 Subscribing to Web Resources

There is another type of service in the Web which is quite the opposite to the afore mentioned approaches in terms of the data flow. It is the concept of push notifications on updates and it is an active research area. There are some manifestations of this model for browser-server communication, such as Comet [15] or Server-Sent Events[1]. Webhooks are a method that enables the asynchronous delivery of data whenever it gets available, compared to the need of actively requesting a service to deliver it. They are uniform resource identifiers (URI), which point to a service in the Web, which accepts the data delivered to it. Within the publish/subscribe paradigm, such asynchronous delivery of data is referred to as events, since that's what the appearance of new data is. Webhooks are callbacks that can be placed by a Web App provider or a user at a remote location, informing the data proliferating site about their interest in the data. Both parties are services in the Web, since the Webhook providers accept the data delivered to their URI and the Webhook recipients offer to send the data. PubSubHubbub[2] an open server-to-server publish/subscribe protocol that uses webhooks for servers to address their interest in updates from other servers. Only through such push notifications a reactive system can truly be reactive in the sense of event-drivenness.

2.1.5 Towards Simple Access and Communication

With JavaScript's success as browser scripting language and recently also as server-side programming language, JavaScript Object Notification (JSON) as an alternative to XML has become popular for data representation throughout the Web. It is also because of its human-readable format and

[1]http://dev.w3.org/html5/eventsource/
[2]http://code.google.com/p/pubsubhubbub/

often simple parsing into data structures of existing programming languages. There is a notable trend towards RESTful services in the Web that offer JSON communication. They benefit from simple but powerful interfaces and easy to debug human-readable communication, which eases integration into other applications, along with the reduced communication volume. Together with client- and server-side Web APIs the Web becomes ever more programmable.

2.2 Reactivity through Event-Condition-Action Rules

In this chapter we have so far shown research in different areas that lead towards a programmable Web. As a result of this research, it is getting easier to compose and orchestrate services in the Web, but reactivity needs to be programmed specifically by experts and general approaches are only available in specific domains. Several studies [2][11][12][25][26] have been made on reactivity. They point out Event-Conditon-Action (ECA) rules as the most adequate way to impose reactivity on a system. As the name already suggests it bases on an event-driven architecture (EDA) and ECA rules consist of three parts:

- Event: An event identifier, that enables detection of a triggered event
- Condition: Expressions to be evaluated to determine whether an action is triggered
- Action: A set of instructions that complete the reactive behaviour

Several different rule languages have been developed for different domains. We will give a brief overview over the research done that relates to our goal, reactivity in the Web. During our research, apart from identifying the key properties of different rule languages, we analyzed them with respect to a certain use case, in order to determine their applicability for our research goal. The use case's ECA rule is:

- Event: Receipt of an Email
- Condition: Check for a certain sender
- Action: Store it remotely via a Web API

We also tried to get access to existing rule engines for each rule language, since we aim to build our model as well as our own reference implementation on top of existing work.

2.2.1 Rule Languages & Rule Engines

The Resource Description Framework (RDF) is a collection of specifications to model informations in the Semantic Web [6]. Papamarkos et al. (2004) published an ECA language for RDF: RDF Triggering Language (RDFTL). It was designed to react on insert and delete events within RDF repositories and for an action notify users and propagate the changes through related resources. RDFTL bases on RDF resources which need to run engines. These engines retrieve events, detect changes and communicate them as events to other engines and actions are executed on local repositories. Through distributed engines, RDF resources can be made reactive. We envision an engine that orchestrates the Web, rather than relying on other Web sites to incorporate our model. But still their research provides important insights on reactivity through ECA rules.

The rule language XChange [27] emerged from the Reasoning on the Web with Rules and Semantics (REWERSE) project [34], which took place from 2004 to 2008. It was designed to track changes in dynamic web resources and add reactive behaviour in a way that such changes influence other dynamic resources. XChange incorporates the vision of distributed, event exchanging, rule engines. Those rule engines execute actions on local data or issue new events. The local-only actions oppose our vision to orchestrate heterogeneous Web resources through reactive behaviour. Eventually a dedicated XChange rule engine could be realized, which is enhanced to translate XChange actions into communication with remote Web resources. The use case applicability study was promising but access to a reference implementation of an engine, in order to enhance it with our vision, could not be gained. Still the thorough research done with the language XChange holds valuable concepts, especially in terms of temporal event composition.

JSON Rules [19] was introduced 2008 as a language to react on specific Document Object Model (DOM) tree states of a webpage and as reactive behaviour control the browser and also DOM again. The incorporation of script function calls into the action part of the language allows the abstraction of eventually complex action behaviour. This feature influenced our concept as it allows for different levels of complexity to be offered as the reactive system is growing. JSON Rules is bound to DOM tree events and actions, where we aim to react on any events happening in Web Resources and also execute actions on them.

The Rule Markup Language (RuleML) [8] is a language written in XML and aims to standardize many different types of rules. Reaction RuleML [26] is an enhancement of the existing standard by reactive rules. Reaction RuleML subsumes:

- Complex Event Processing (CEP)
- Knowledge Representation (KR) calculi
- Event-Condition-Action (ECA) rules
- Production (CA) rules
- Trigger (EA) rules

Reaction RuleML represents thorough research for a language to describe virtually any type of reactivity. Together with the expression in XML it doesn't score with readability, but provides a way to define a multitude of rule types and the interchangeability of them between different sites in the Web. Since our vision does not require interchangeable rules, we chose an internal JSON representation for our rules to have human-readability, simple parsing and efficient storage. A system that relies on RuleML is Rule Responder [**?**]. Rule Responder connects different types of heterogeneous rule engines together over the Mule open-source Enterprise Service Bus (ESB) which acts as a communication middleware to exchange rules expressed in RuleML.

A recent research outcome (Windley, 2011) was the Kinetics Rule Language (KRL) [38] together with the Kinetic Rules Engine (KRE). It was invented to impose reactivity to the Web and incorporates many different event origins and action resources. The language is based on declarative syntax, enriched with imperative elements. A handy feature are the ways to activate webpages, which bridges the gap between the user's browser and the centralized KRE. Either a user can install a browser plugin which will communicate with the KRE, or a webpage provider can include a library in order to get events from accesses to the webpage. Through this events can be raised from the browser and actions can execute in it. The KRL fits very well into our concept and only a few reasons kept us from realzing our reference implementation on top of the KRE, such as:

- complexity required to maintain states with a declarative syntax
- system footprint of the KRE
- Perl, a procedural programming language, as base of the KRE

The concept of the KRL is promising in terms of orchestrating the Web through reactivity. But we decided to implement a light weighted reference system, using an event-driven programming language that is laid out for an event-driven architecture.

Table 2.1 gives an overview of the key properties of existing rule languages and their key properties for our research:

- Event Origin: Resources type from where the events originate.
- Distributed: Whether the language is laid out to run on a centralized or distributed architecture. All examined rule languages that support distributed architectures can as well run on a centralized architecture.
- Action Resource: Resource type on which actions are executed.
- Accessible Engine: Determines whether a reference implementation of an engine was accessible.
- Applicability to our concept: Names the main difference to our envisioned concept.

Rule languages that support a distributed architecture require engines to be distributed on sites that should be reactive. This is not service-oriented and does not regard the Web's heterogeneous nature and the need to access such services.

Other examined Rule Engines were the Object-Oriented Java Deductive Reasoning Engine for the Web (OO jDrew), Prova, and Drools Fusion. They are all implemented in Java and have their own rules syntax which is more or less closely related to Java, with inline Java code. Some of them have a heavy system footprint because they base on existing frameworks, such as Drools Fusion which requires a Java graphical user interface or communication middlewares. Our decision not to use any of these existing approaches for a reference implementation was, because all of them only provide limited support for our concept or have a heavy system footprint, caused by the communication layer such as the JBoss ESB. We envision a scalable event-driven system from its deepest roots that allows the orchestration of heterogeneous Web resources. These resources are already available and accessible and we do not need to alter them in order to impose reactivity to the Web. Such a system does not require a messaging middleware because the Web itself is the communication channel to receive events and execute actions.

2.2.2 Event Composition

An important research area in event-driven architectures is event composition, or Complex Event Processing (CEP). Event Composition is the research for methods to get events into relations and also to identify pattern among them. It [1][3][18][24][30][33]

CEP is a superset of ESP

Language	**Event Origin**	**Distributed**	**Action Resource**	**Accessible Engine**	**Applicability to our concept**
RDFTL	RDF Repository Changes	Yes	(Local) RDF Repository	-	Only Web sites with engines are reactive
XChange	Web Resources	Yes	Local Resources	-	Actions in remote Web Resources missing
JSON Rules	DOM Events	No	Browser / DOM	-	Only Browser / DOM Events
RuleML	Web Resources	Yes	Local Resource	(OO) jDrew, Prova, Rule Responder	Complex Syntax
KRL	Web Resources	No	Local & Remote Web Resources	KRE	User-specific Web App functionality missing

Table 2.1: Key Properties of existing Rule Languages with respect to our Concept

Chapter 3

Conceptual Model for Reactive Information Systems and their Services

In the previous chapter we have shown that reactivity and the evolution of Web Resources as data and functionality providers have received a great deal of attention. Our goal is to combine both research fields in order to impose reactivity to our information space, the Web Resources.

3.1 From Real Events to Events in the Web & their Composition

Real events are bound to a spatial location and a point on the time axis. An earthquake for example always has an epicentre and an occurrence time.

Different points on earth's surface would feel the earthquake, which originates from the same epicentre, at a different point in time with a different intensity. A Web event model of an earthquake would consist of a large number of identical **ground-shake** events that occur at different points in time and places. Therefore they would hold different spatial location informations and intensities. These events can be thought of as emitted into the Web by a seismometer sitting at the corresponding location. Within the Web, events lose their tight

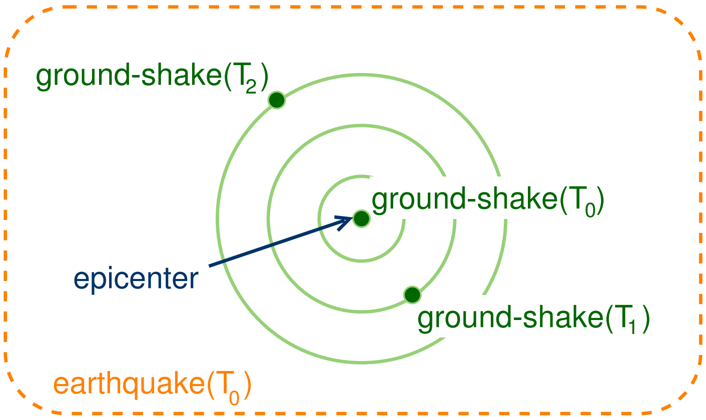

Figure 3.1: Web Event Model of an Earthquake

coupling to locations and retain only a time component. The event instances keep this information as descriptive metadata. A reactive system such as we envision it, could detect these **ground-shake** events and react on behalf of each one of them. Because of the Web's latency these events do most certainly not arrive at systems within the Web in the original order, in which they were triggered. They also do most likely not arrive in the exact same order for all systems. This leaves us with time as the only important factor left, to distinguish events from each other in the first place. To get an earthquake event out of all these ground shaking informations floating through the Web, somebody would need a reactive system that detects these events and assembles them into one earthquake event, together with a computed epicentre and magnitude. Such a system (we call it **earthquake-tracker**) would own an earthquake model that allows it to decide whether a **ground-shake** event belongs to one physical earthquake or to another one, depending on its spatial location information and the intensity at that point in time. It could then emit a more complex **earthquake** event (with epicentre and magnitude) that allows other systems to interpret this physical event and react on behalf of it.

Let's take another system that reacts on a physical earthquake. It is now left with a multitude of different options on how to react. It could only react on the **earthquake** event which is coming from the system above (**earthquake-tracker**) that applies its earthquake model to the incoming **ground-shake** events. But how long will it take for this system to deploy its **earthquake** event? Eventually it waits for one round-trip of a seismic wave around the world, which takes approximately half an hour. What if it waits two or three round-trip times in order to collect more accurate data? And what if our new system wants to react as fast as possible in order to warn people all around the world. It would then need to react on a small subset of the **ground-shake** events in order to quickly identify a real earthquake and take measurements, e.g. immediately send out text messages to people, or to deploy yet another (this time **earthquake-alert**) event into the Web's information space. This relatively simple example discloses the complex nature of event-driven systems, but also their high flexibility and fine grained tuning possibilities.

3.2 Web Resource's Event & Action Space

For a conceptual model, the information space in which the events are triggered and the actions are invoked, needs to be identified. During our research we encountered many different event or action providing subsets of the Web that can be incorporated into our model:

- World Wide Web
- Web of Things
- Services in the Web

We have already shown in chapter "Related Work", that there are basically two different ways how the Web's information space is accessible, i.e. either functionality and data have to be requested, or data is pushed through Webhooks. All of the above listed information space subsets require at least one of these two access methodologies. And through these access methodologies are we able to turn the information space of the Web into events and actions. The World Wide Web [5], as envisioned by Tim Berners-Lee, is an information universe of interlinked documents, that a user can browse through. In our model, we can pull events directly from the World Wide Web. For example, most documents in the World Wide Web are subject to changes and such changes can be translated into events.

We gave an introduction into Services in the Web in chapter "Related Work"

Either there is an event producer which proactively pushes events into the Web, or a service is offered whose responses can be turned into events. These events and actions can have a solely virtual nature or, in the case of data from the "Web of Things", also a physical nature. A virtual nature can be anything from a static webpage to offered services, such as a detected change on a webpage can become an event as well as a service answer is interpreted as such for example a new mail arriving. A physical nature for events could for example be measurements from a rain detector, an action could be a window shutting automatically.

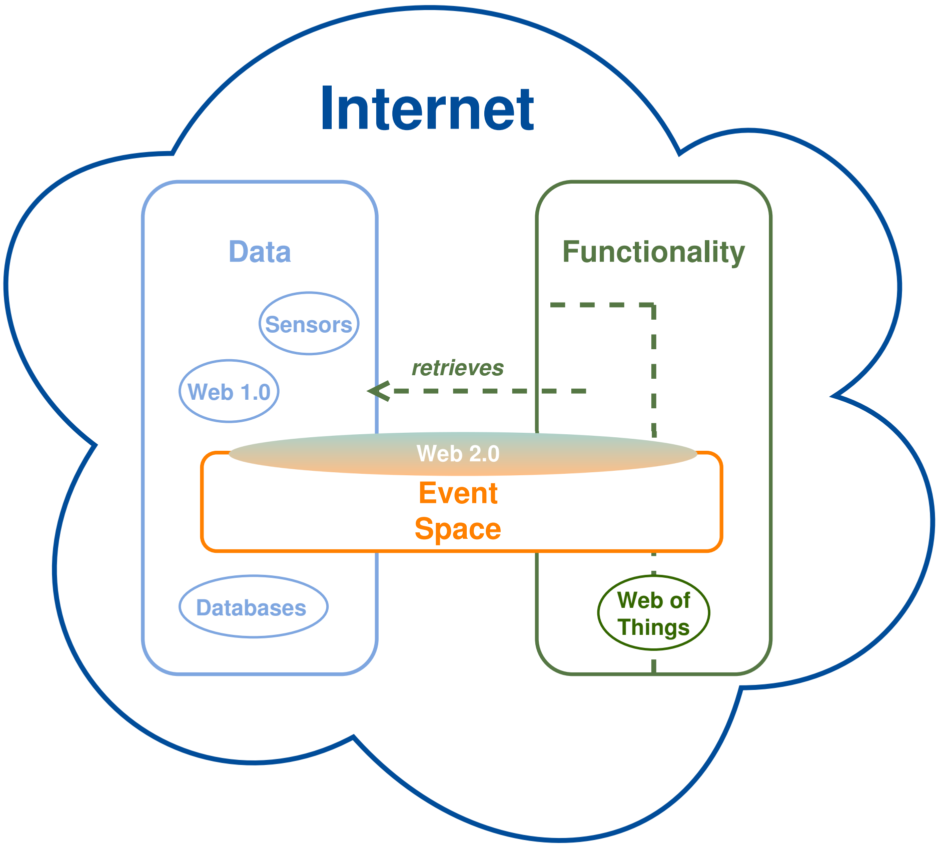

Figure 3.2: The Internet as Information Space

There are different categories of events and also different ways how they can get into the Web:

Actions

- Event Redirection
- Event Enrichment
- WebApp Actions

3.3 Impose Reactivity to the Web Information Space

In the last section we showed how mashups create additional value for the Web by combining several Web API's. But it turned out, that such mashups are closed systems, which often only allow little degree of parametrization. To get past such limitations and define a conceptual model for reactive Web systems, it is necessary to define a existing rule languages, rule engines,

Existing ECA systems all act on local data. Looking at (Wikipedia...) their definition is actions on local data. This does only add reactivity to these systems and not to the Web per se.

Such systems are merely event sinks which add fairly any value to the Web, except for the individual users and the system itself.

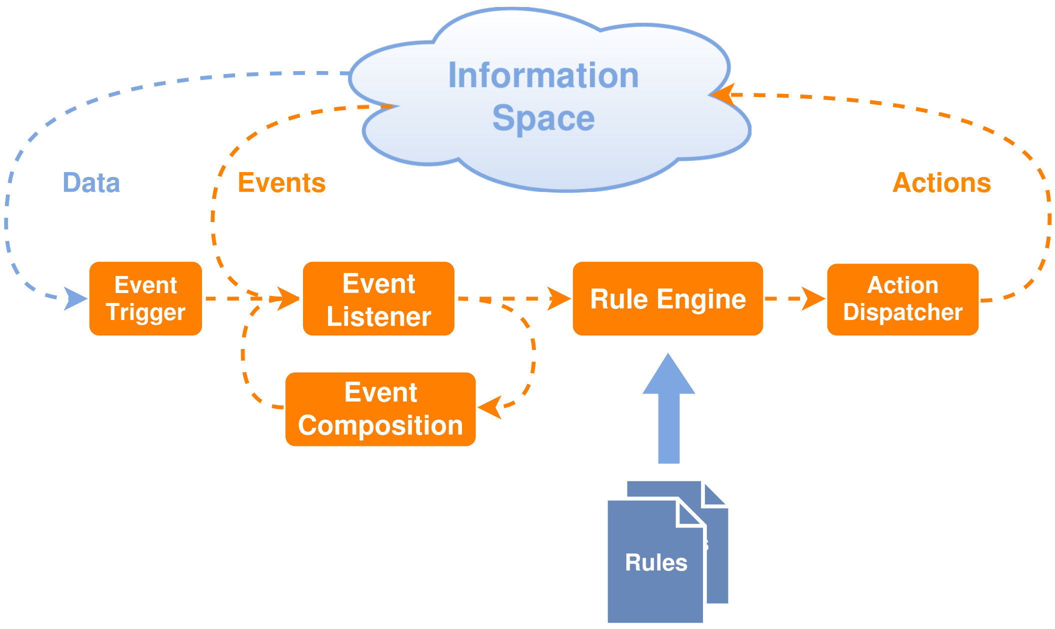

Figure 3.3: Conceptual Model for Reactive Information Systems and their Services

3.4 Rule Language for Reactive Information Systems

Because of the heterogenous nature of the Web actions need to be abstracted as long as we do not limit ourselves e.g for RESTful access to services

Chapter 4

Use Cases for Reactive Information Systems

We have introduced a conceptual model for reactive information systems and its services. In this chapter we will show how this model applies to certain use cases.

4.1 Reacting on changes in the World-Wide Web

A huge amount of information is accessible through the World-Wide Web. Many documents within it are dynamic in the way that they change over time. Some might change in an interval of a few minutes (e.g. news), while others change much slower such as konwledge bases. To detect such changes a site in the Web needs to keep track of the document history and trigger events if there is a change. A lot of research has been mad For a general approach such a site would require a query language for web resources that is able to detect changes.

4.2 Augment existing Web Applications

Web applications, such as webmails, social networks or content management systems (CMS) are widely spread and used by a large number of internet users. But often users or developers miss some features or interoperability with other web applications, which would result in augmented functionality and also in less work. Features of that kind could also include data and functionality from other sites on the web. This would require web applications to mashup together and to grab data or invoke functionality on each other. Many kinds of augmentations will not be implemented by the web applications themselves because they are very specific to a small number of their users. With a reactive information system, users and developers could realize such features on their own.

4.2.1 Enrich CMS Posts with Additional Information

Every new post to a content management system can be modeled as an event. In the case that a user would like to enrich such a system with founded knowledge from a remote resource, reactivity in the Web can be used to enrich it. Augmenting an existing CMS can be realized by a rule that evaluates new posts and checks whether knowledge tags are included in the post.

Whenever there are tags included within the post, the reactive system will enrich the post with additional knowledge to these tags from a remote service of the user's choice.

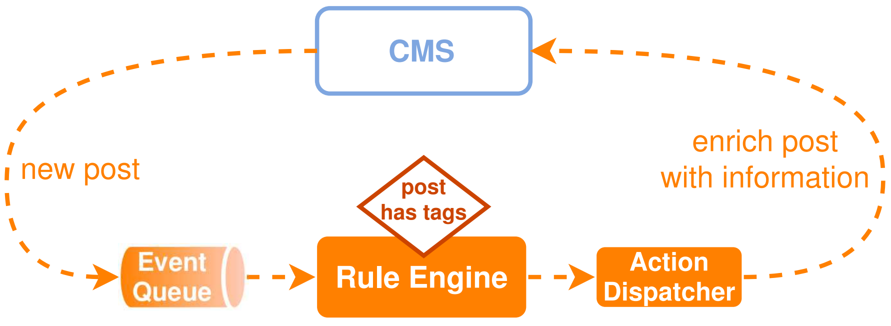

Figure 4.1: Enrich CMS Post with Remote Knowledge Data

4.2.2 Workflow Automation

Within such web applications, users often have very specific workflows. And because workflows always start with an event, they are predestined to be automated by a reactive information system. As an example for workflow automation, course and student exercise submission administration can be taken care of by a reactive information system. Whenever a new semester starts, the reactive system will detect this through one of its rules and command an action dispatcher to set up infrastructures for courses. This can also include grabbing course data from an official web page and including it into the infrastructure, thus eliminating the need for manual data copy tasks.

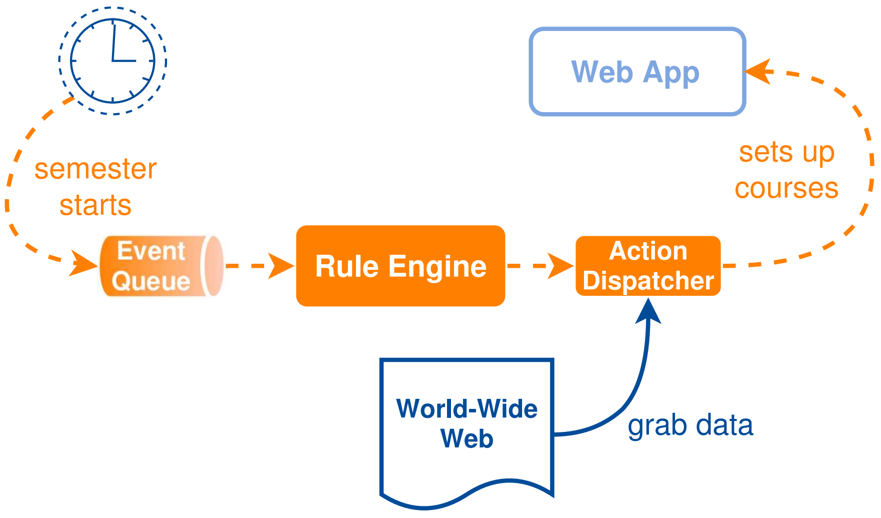

Figure 4.2: Create course resources at semester start

After setting up the semester courses, the reactive system is ready to process new student registrations for these courses. It automatically associates students into the afore mentioned infrastructure and sets up additional infrastructure such as an exercise submission container. Whenever a course tutor submits a new exercise to the course resource, the system will detect this and spread this information to the students, together with a deadline. The students are expected to submit their exercise solutions before the deadline to the exercise submission container that was created reactively for them.

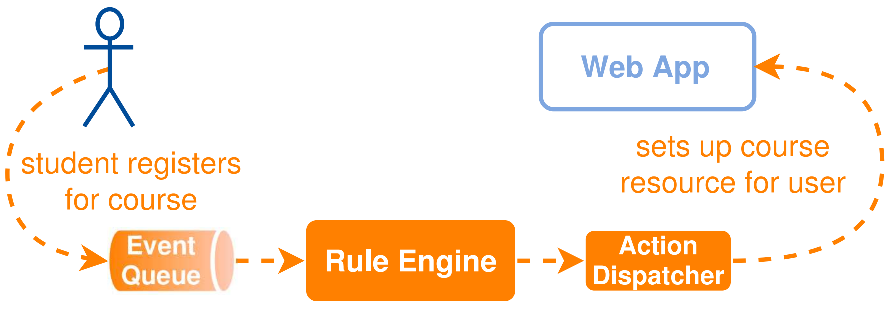
Web App
student registers for course
sets up course resource for user
Event Queue
Rule Engine
Action Dispatcher

Figure 4.3: Create course resource for registered student

A certain amount of time before the deadline, e.g. one day, the reactive system will detect the deadline and process events that depict the current exercise submission status per student. If the system detects a student who hasn't uploaded her exercises yet, it will notify her about the deadline. This is an additional service that gives students the chance to react on a missed exercise submission deadline. As soon as the deadline passed, the system will revoke write-rights to the exercise submission container and therefore disallow submissions which are too late.

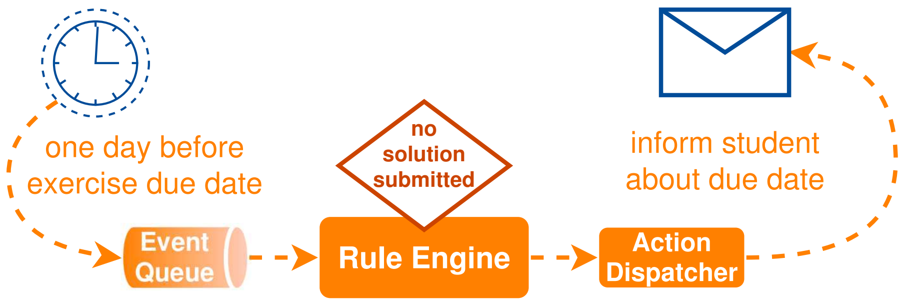
one day before exercise due date
no solution submitted
inform student about due date
Event Queue
Rule Engine
Action Dispatcher

Figure 4.4: Notify student before exercise due date

4.3 Service Functionality and Availability Checking

Services offered through the Web aren't monitored or tested by users or developers from other sites. If they rely on correct functionality or availability they need a way to assert this. It is also possible that an owner of such a service doesn't have the tools to monitor his own services Whenever such a service isn't working correctly anymore or stops responding, these users or developers need to be able to react on this before it is too late. With a reactive rule in place that evaluates Service Testing results, measurements can be taken early. One action to such a failing service test could be an automatic switching of the utilized service within an action dispatcher, so that from then on it uses one which still works correctly.

4.4 Exploiting the Web of Things

A model for reactive information systems becomes also very interesting in the context of the Web of Things. Through the small connected devices, a lot of sensor data become accessible via the Web and can be used as events to trigger actions. These actions could also be part of the Web of things, if there are Things that offer services. One example of a reactive rule, that

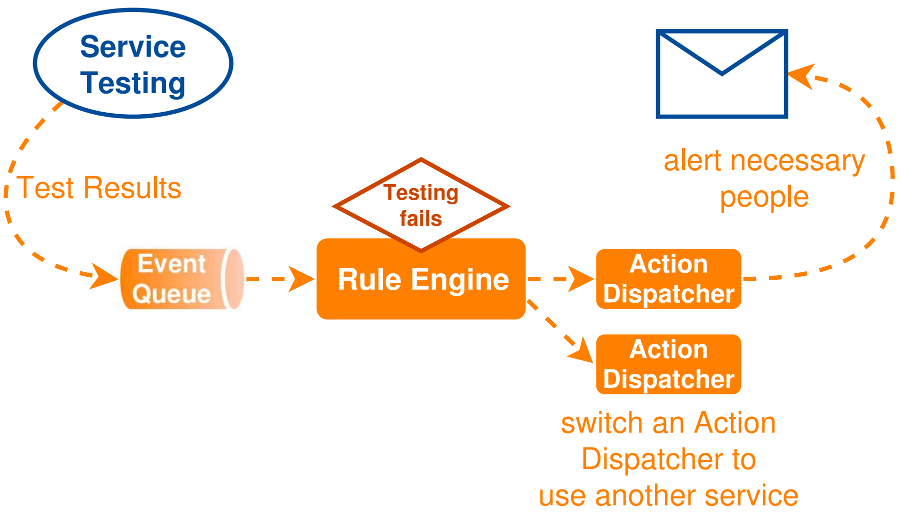

Figure 4.5: Test proper Service Functionality and Availability

has parts in the Web of Things, is that of a server room which has a defective cooling. The increasing temperature eventually causes the servers to shutdown or even fail. Servers in this room could push current state information into a reactive system. The reactive system could take measurements if it detects a certain pattern that will lead to an overheating of all systems. It could inform certain (not so important) servers to gracefully shutdown and additionally inform administrators, who otherwise might miss the shutdown. Even better would be if it would have the power to kick in an additional emergency cooling system to prevent the shutdown of any of the servers.

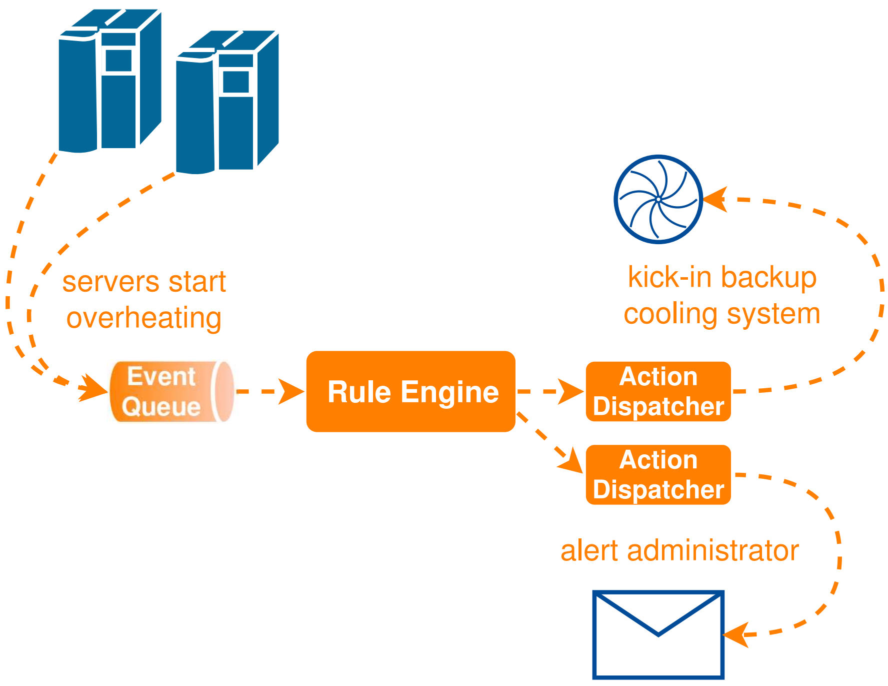

Figure 4.6: Measurements on Server Failure

Another scenario gets more realistic with the increasing number of homes that are connected to the Web. A home or apartment owner has his light controls attached to the Web. The first thing a reactive system could do, is that it detects holidays in the owner's agenda and automatically sets the light control to somewhat reasonable random during his absence. This would make suspicious characters, which are eventually interested in his wealth, think that he's still at home. In combination with another Thing that is connected to the Web and always accompanies people, the cell phone, an even more interesting application scenario can be thought of. The phone would push location information about the owner into the system. Whenever the owner gets close to his house, the reactive system could turn on the light in the entry area, pull up some nice music. On a Wednesday evening it could also inform the delivery service that they can deliver the owner's preferred dish now, because the owner is doing this always on a Wednesday evening.

Chapter 5

Prototype System

The prototype system is the realisation of our conceptual model for reactive Information Systems and their services. During our research we identified asynchronous communication and event-driven architecture (EDA) as necessary key properties for a reference implementation of our conceptual model. Therefore we decided to rely on the recent adoption of JavaScript to application development, which is made possible through Node.js. We believe in the future success of JavaScript for application development because the key concepts of asynchronous communication and non-blocking I/O are important properties for todays large scale applications. Another advantage of this technology is the human-readable JavaScript Object Notation (JSON) for data exchange. JSON builds only on two data structures:

- Object: An unordered collection of name/value pairs, which can also be implemented as a hash map, dictionary or struct.
- Array: An ordered list of values, which can also be implemented as a record, vector or list.

A value can be an object or an array, but also a unicode string, a number, boolean or null. This allows for any arbitrary depth and chaining of the supported data representations. Since all data within JavaScript programming code is already in JSON format, it can easily be marshalled into one string and communicated to other applications without overhead. Since JSON can be implemented in virtually every programming language, it received a lot of attention and is supported by many modern Web resources.

The prototype consists of a queue which receives all incoming events, and an engine that picks the events from the end of the queue whenever it is idle. The engine checks the event against its stored ECA rules and fires the actions whenever a rule applies to an event. When a new rule is stored, the engine instantiates the action modules that are required in case an event arrives at the system which triggers the actions. The Web is a heterogeneous resource with many different types of systems and services, thus we can't rely on them for providing events to the webhooks of our system. Thus we also implement the so called "event triggers" a way to pull events into our system through. All they do is checking Web resources for changes and push events into the system whenever they detect one.

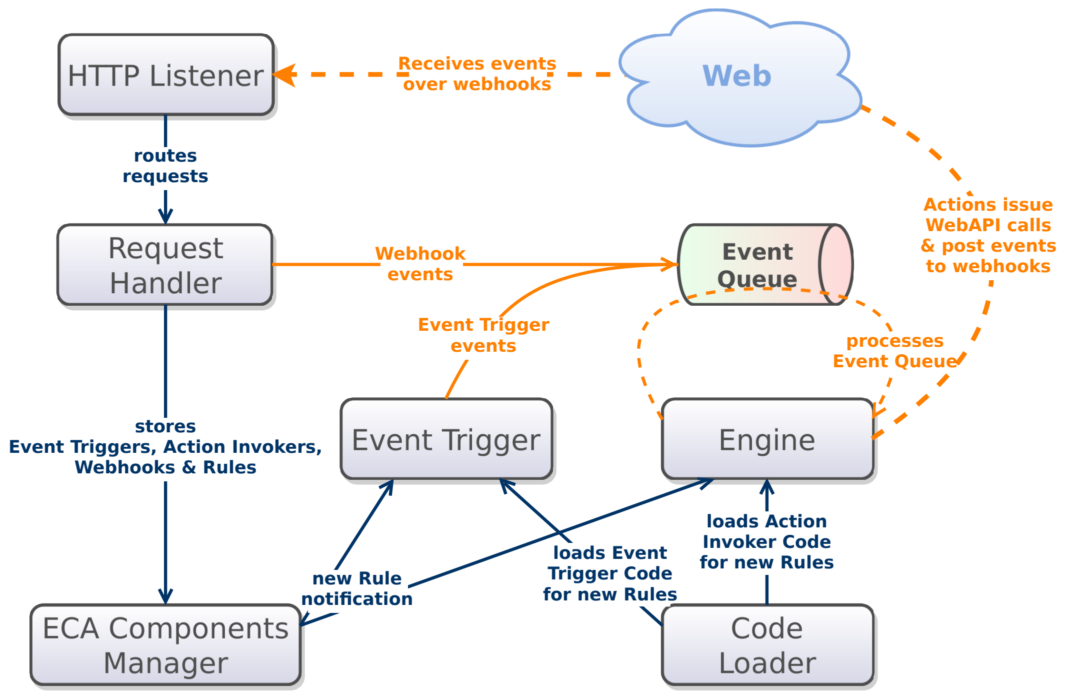
HTTP Listener
Receives events over webhooks
Web
routes requests
Request Handler
Webhook events
Event Queue
Actions issue WebAPI calls & post events to webhooks
Event Trigger events
processes Event Queue
stores Event Triggers, Action Invokers, Webhooks & Rules
Event Trigger
Engine
loads Action Invoker Code for new Rules
new Rule notification
loads Event Trigger Code for new Rules
ECA Components Manager
Code Loader

Figure 5.1: Prototype Process diagram

5.1 Event Trigger

5.1.1 Polling

5.1.2 Webhooks

5.2 Action Dispatcher

5.3 ECA Rules in the Engine

During our research we found a troublesome server room that shows how the Web of Things can be exposed through our model. This server room suffered from a defective cooling system which lead to a drastical increase of temperature in certain circumstances. As a consequence certain server automatically shut themselves down as safety measurements. Eventually, these shutdowns weren't detected immediately by the people that administered these servers, therefore unnecessary downtimes were the result. As a very quick fix to inform certain administrators about the shutdown of their server, we started pinging these servers and pushed the results int

5.3.1 Example Use Cases

5.4 Web Programming

5.4.1 Node.js

5.4.2 Callback Functions & Asynchronous Closures

Often, optimization approaches and programming language concepts require special attention to avoid common pitfalls. When closures are used as asynchronous functions, developers need to be very careful not to end up with race conditions.

Looking at an example of sequential code execution in Figure 5.2, we see that function execution of `fA` is halted until function `fB` is finished. If `fB` happens to be a latency-driven I/O operation the completion of `fA` could be deferred for a relatively long time. While the application waits for the completion of the I/O operation, some remaining operations in `fA` could eventually already be executed without causing any race conditions.

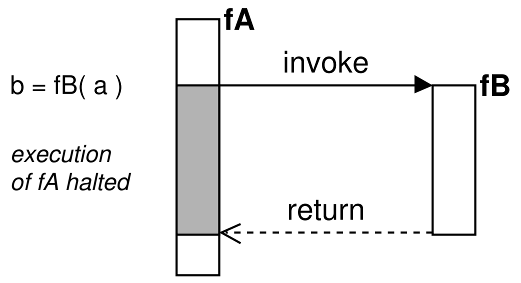

Figure 5.2: Synchronous Function Call

Asynchronous code execution, as shown in Figure 5.3, allows non-blocking and thus scalable applications. Non-blocking operations are a remedy for optimzed resource allocation and open up ways to overcome previously described unnecessary resource bindings. Processing any kind of latency-driven I/O operation asynchronously (e.g. filesystem access and socket communication) exploits resources that would otherwise be bound while waiting for completion. Such operations are processed and completed whenever required resources are available.

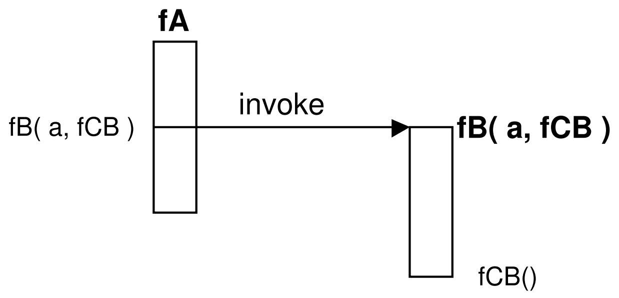

Figure 5.3: Asynchronous Function Call

Often other operations depend on the completion of asynchronous operations, hence their execution needs to be deferred. This necessary code execution deferral is achieved through the use of callback functions, denoted `fCB` in Figure 5.3. Any code placed in a callback function, which is assigned to an asynchronous operation, is only executed after the respective asynchronous operation completed. This allows stacking of functions and operations upon each other which automatically results in a flexible and event-driven application.

So far we didn't regard the context for such asynchronous functions. If a function has access to the enclosing context where it was invoked in, it is called a closure. Closures play an important role in ECMAScript[16], which is the base for widely-spread script languages like JavaScript, JScript and ActionScript. Closures in ECMAScript[16] are defined such as they have access to the context of the function they were created in. This is shown in Figure 5.4 where c from `fA`'s context is accessible from within `fB`, assuming that `fB` was created in `fA` and not only invoked from there. Closures make it necessary for the context of the outer function to survive past its execution so no references are broken. This is depicted through the "extended context lifetime" in Figure 5.4. Using asynchronous closures it becomes evident, that the context in the invoking function can change while the closure is still computing and eventually referencing the outer context, thus causing race conditions. This will be most obvious in a loop that immediately invokes `fB` several times, as shown in Figure 5.5. In such a setup c will have different values in the same part of different invocations of `fB`.

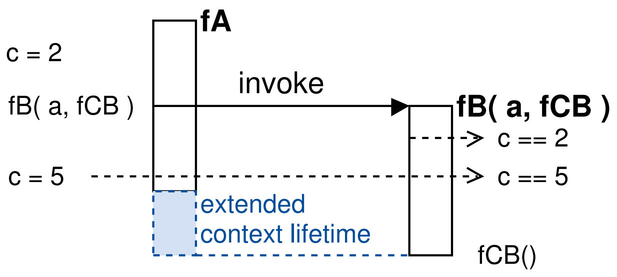

Figure 5.4: Closure Scope and referenced context

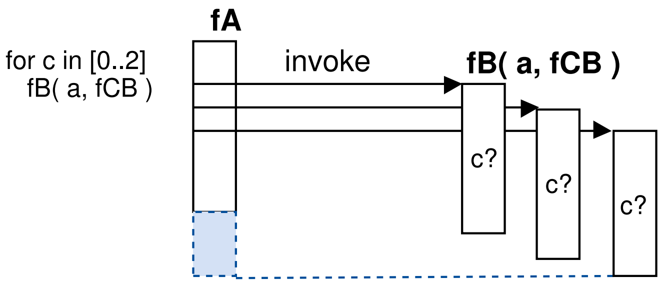

Figure 5.5: Closure context changes in a loop

Those event-driven context overwrites can be taken care of by shielding the closure from context changes, as shown in Figure 5.6. To shield the closure form context changes, closure `fB` needs to create another closure `fC` and return it to `fA`. The argument passed to `fB` is the context (c in Figure 5.6) that might change but requires to be persistent for one invocation. `fC` has now c as a fixed context, which can't be overwritten anymore. Now the only thing left is `fC` needs to be invoked and it will retain the original context. This implementation is necessary when the closure acts as a callback function for asynchronous operations, to preserve the original context in case it is required within the callback function.

An example of how closure contexts can be shielded is shown in the Listing 5.1.

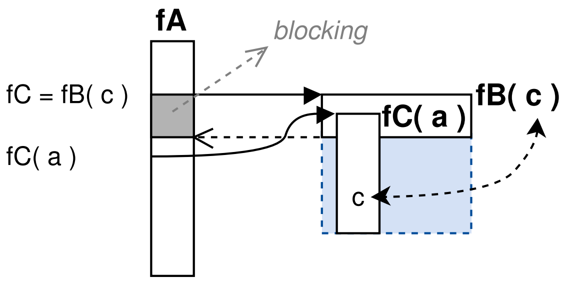

Figure 5.6: Closure Context Shielding

```
var fB = function( c ) { // Declare a function...
  var fC =  function( a ) { // ( <-- function to return )
    console.log( c );
  };
  return fC;
};
for( var c = 0; c < 100; c++ ) {
  // ... before you assign it to an event happening in the future:
  var fC = fB( c );
  setTimeout( fC, 3000); // will be executed after the loop ended
}
```

Listing 5.1: JavaScript Context Shielding

Chapter 6

Conclusions & Future Work

Bibliography

[1] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul, Plan-based complex event detection across distributed sources. *Proceedings of the VLDB Endowment*, 1(1):66–77, 2008.

[2] JoséJúlio Alferes and Ricardo Amador. r 3– A Foundational Ontology for Reactive Rules. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 933–952. Springer Berlin Heidelberg, 2007.

[3] Darko Anicic, Paul Fodor, Sebastian Rudolph, Roland Stühmer, Nenad Stojanovic, and Rudi Studer. A Rule-Based Language for Complex Event Processing and Reasoning. In Thomas Lukasiewicz Pascal Hitzler, editor, *Web Reasoning and Rule Systems - Fourth International Conference*, volume 6333 of *LNCS*, pages 42–57. Springer, September 2010.

[4] Alistair P. Barros and Marlon Dumas, The Rise of Web Service Ecosystems. *IT Professional*, 8(5):31–37, 2006.

[5] Tim Berners-Lee, Robert Cailliau, Jean-François Groff, and Bernd Pollermann, World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.

[6] Tim Berners-Lee, James Hendler, Ora Lassila, et al., The semantic web. *Scientific american*, 284(5):28–37, 2001.

[7] Andrew D. Birrell and Bruce Jay Nelson, Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[8] Harold Boley. The RuleML family of web rule languages. In *Principles and Practice of Semantic Web Reasoning*, pages 1–17. Springer, 2006.

[9] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (SOAP) 1.1, 2000.

[10] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau, Extensible markup language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[11] Francois Bry and Paula lavinia Patranjan, Reactivity on the Web: Paradigms and Applications of the Language XChange. *J. of Web Engineering*, 5:2006, 2005.

[12] François Bry and Michael Eckert. Twelve Theses on Reactive Rules for the Web. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *Current Trends in Database Technology – EDBT 2006*, volume 4254 of *Lecture Notes in Computer Science*, pages 842–854. Springer Berlin Heidelberg, 2006.

[13] Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci. DashMash: A Mashup Environment for End User Development. In Sören Auer, Oscar Díaz, and GeorgeA. Papadopoulos, editors, *Web Engineering*, volume 6757 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin Heidelberg, 2011.

[14] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.

[15] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Consistency and scalability in event notification for embedded Web applications. In *Web Systems Evolution (WSE), 2009 11th IEEE International Symposium on*, pages 89–98, Sept 2009.

[16] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011.

[17] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[18] N. H. Gehani and H. V. Jagadish. Composite event specification in active databases: Model and implementation. pages 327–338, 1992.

[19] Adrian Giurca and Emilian Pascalau, Json rules. *Proceedings of the Proceedings of 4th Knowledge Engineering and Software Engineering, KESE*, 425:7–18, 2008.

[20] Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.

[21] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010*, pages 1–8, Nov 2010.

[22] Keman Huang, Yushun Fan, and Wei Tan. An Empirical Study of Programmable Web: A Network Analysis on a Service-Mashup System. In Carole A. Goble, Peter P. Chen, and Jia Zhang, editors, *ICWS*, pages 552–559. IEEE, 2012.

[23] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards Service Composition Based on Mashup. In *Services, 2007 IEEE Congress on*, pages 332–339, July 2007.

[24] Gero Mühl, Ludger Fiege, and Peter Pietzuch. Composite Events. In *Distributed Event-Based Systems*, pages 231–251. Springer Berlin Heidelberg, 2006.

[25] George Papamarkos, Alexandra Poulovassilis, and Peter T Wood. RDFTL: An event-condition-action language for RDF. In *Proc. of the 3rd International Workshop on Web Dynamics*, 2004.

[26] Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian, and Tara Athan. Reaction RuleML 1.0: Standardized Semantic Reaction Rules. In Antonis Bikakis and Adrian Giurca, editors, *Rules on the Web: Research and Applications*, volume 7438 of *Lecture Notes in Computer Science*, pages 100–119. Springer Berlin Heidelberg, 2012.

[27] Paula-lavinia Patranjan. *The Language XChange*. PhD thesis, Ludwig-Maximilians-Universität München, 2005.

[28] C. Peltz, Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct 2003.

[29] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[30] P.R. Pietzuch, B. Shand, and J. Bacon, Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, Jan 2004.

[31] ProgrammableWeb: APIs, mashups and code. Because the world's your programmable oyster. `http://www.programmableweb.com`. Accessed: 2014-05-02.

[32] Paul Rademacher. HousingMaps. `http://www.housingmaps.com/`, 2005. Accessed: 2014-6-6.

[33] Setareh Rafatirad, Amarnath Gupta, and Ramesh Jain. Event Composition Operators: ECO. In *Proceedings of the 1st ACM International Workshop on Events in Multimedia*, EiMM '09, pages 65–72, New York, NY, USA, 2009. ACM.

[34] REWERSE - Reasoning on the Web with Rules and Semantics. `http://rewerse.net`. Accessed: 2014-05-08.

[35] Sven Rizzotti and Helmar Burkhart. useKit: A Step Towards the Executable Web 3.0. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1175–1176, New York, NY, USA, 2010. ACM.

[36] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A remote procedure call API for Grid computing. In *Grid Computing—GRID 2002*, pages 274–278. Springer, 2002.

[37] Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma, Discovering how end-user programmers and their communities use public repositories: A study on Yahoo! Pipes. *Information and Software Technology*, 55(7):1289 – 1303, 2013.

[38] P. Windley. *The Live Web: Building Event-Based Connections in the Cloud*. Cengage Learning PTR, 2011.

List of Figures

List of Tables

Listings

Appendices

Appendix A

Rule Languages

A.1 Example JSON Event for Rule Languages

```
{
  "eventname": "email",
  "body": {
    "sender": "sender@mail.com",
    "subject": "Important subject!",
    "textbody": "Hi User,\n\nThis is a lengthy mail body"
  }
}
```

A.2 E-Mail Example Rule expressed in RDFTL

```
ON INSERT document("inbound_queue.xml")/mails/mail
IF $delta/sender[.="sender@mail.com"]
DO DELETE document("inbound_queue.xml")/mails/mail;
  LET $api = resource("www.webapi.com") IN
  INSERT ($api, newcontent,
    <content>New mail: {$delta/subject}</content>)
```

A.3 E-Mail Example Rule expressed in XChange/Xcerpt

A.4 E-Mail Example Rule expressed in Notation 3

A.5 E-Mail Example Rule expressed in JSON Rules

```
{
    "id": 0,
    "conditions": [
        {
            "type": "email",
            "constraints": [
TRANSACTION
  in {
    resource { "http://www.webapi.com"},
    newcontents {{
      insert newcontent { var Mail }
    }}
  }
ON
  xchange:event {{
    xchange:sender { "http://mailserver.com" },
    var Mail -> email {{
      sender { "sender@mail.com" }
    }}
  }}
END
```

```
{ ?x :event "email". ?x :sender "sender@mail.com" }
  => { :webapi :newcontent ?x }
```

```
                {
                    "propertyName": "sender",
                    "operator": "EQ",
                    "restriction": {
                        "type": "String",
                        "value": "sender@mail.com"
                    }
                },
                {
                    "bind": "$S",
                    "propertyName": "subject"
                }
            ]
        }
    ],
    "actions": [
        "webapi('addcontent', $S)"
    ]
}
```

A.6 E-Mail Example Rule expressed in Kinetics Rule Language (KRL)

A.7 E-Mail Example Rule expressed in (Reaction) RuleML

```
<Rule style="active">
  <on>
    <Event>
      <Atom>
        <Rel per="value">mail</Rel>
        <Var>sender</Var>
        <Var>subject</Var>
      </Atom>
```

```
rule store_mail {
  select when mail newmail
  sender re#sender@mail.com#
  subject re#*# setting(subj)
  http:post("http://www.webapi.com/newcontent")
  with params = {
    "text": subj
  }
}
```

Listing A.1: E-Mail Example rule in KRL

```
      </Event>
    </on>
    <if>
      <Atom>
        <op><Rel>equals</Rel></op>
        <Var>sender</Var>
        <Ind>sender@mail.com</Ind>
      </Atom>
    </if>
    <do>
      <Atom>
        <oid><Ind uri="http://webapi.com"/></oid>
        <Rel>newcontent</Rel>
        <Var>subject</Var>
      </Atom>
    </do>
  </Rule>
```

A.8 Prototype Rule transformed into JSON

```
{
  "event": "mail",
  "conditions": [
    { "sender": "sender@mail.com" },
  ],
  "actions": [
    {
      "api": "webapi",
      "method": "newcontent",
      "arguments": {
        "text": "$X.subject"
      }
    }
  ]
}
```

Appendix B

Rules

B.1 Binder Annotations

B.1.1 Binder Annotations

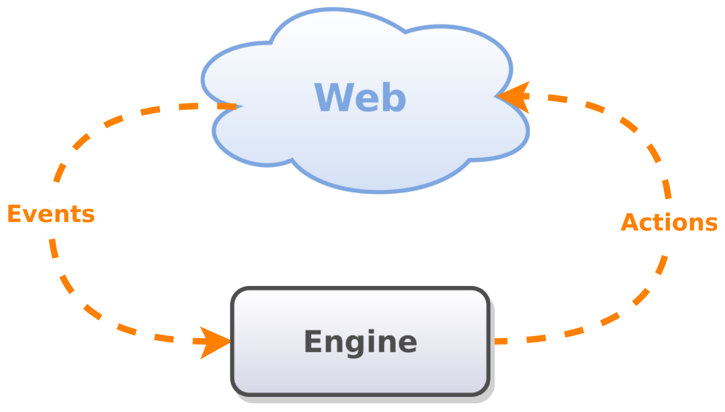

Figure B.1: SHOULD NOT SHOW UP

Appendix C

Benchmarking

C.1 Java

```java
/*
 * BenchmarkingDeferred.java
 */
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.HashMap;

public class BenchmarkingDeferred {

  private static Runtime runtime = Runtime.getRuntime();
  private static final ScheduledExecutorService worker =
    Executors.newSingleThreadScheduledExecutor();

  private static void deferFunctionCall( int numScopeVars, int delay, String scopeId )
    HashMap<String, String> mapVars = new HashMap<String, String>();
    for( int i = 0; i < numScopeVars; i++ ) {
      mapVars.put( "id" + i, "12345678" ); // 8 bytes per stored scope variable
    }
    Object context = new TimeoutContext( "TimeoutFunction" );
    Runnable task = new RunnableCallbackFunction( mapVars, context );
    worker.schedule( task, delay, TimeUnit.SECONDS );
  }

  public static void main( String[] args ) {
    long startTime, stopTime;
    int numVars = 10, firstArg = 0;
    firstArg = Integer.parseInt( args[0] );
    numVars = Integer.parseInt( args[1] );
    int j = 0, numFuncs = 1 << firstArg;

    startTime = System.nanoTime();
    while( j++ < numFuncs ) {
      deferFunctionCall( numVars, numFuncs * 10, numFuncs + "(" + j + ")" );
    }
    stopTime = System.nanoTime();

    // [...] benchmark system out

    worker.shutdownNow();
  }
}

/*
 * RunnableCallbackFunction.java
 */
import java.util.HashMap;

/*
 * The Callback function instance.
 */
public class RunnableCallbackFunction implements Runnable {

  // The hashhmap is used to store variables and their value as the scope
  private HashMap<String, String> mapScope;
  private Object context;

  public RunnableCallbackFunction( HashMap<String, String> scope, Object context ) {
    this.mapScope = scope;
    this.context = context;
  }

  // If this is executing, we didn't wait long enough and the
  // benchmark time is compromised
  public void run() {
    System.out.println( mapScope.toString() );
  }

}

/*
 * TimeoutContext.java
 */
public class TimeoutContext {
  private long idleTimeout = 1;
  private long idlePrev;
  private long idleNext;
  private long idleStart = 140000505;
  private String onTimeout = null;
  private boolean repeat = false;

  public TimeoutContext( String cb ) {
    this.onTimeout = cb;
  }
}
```

Listing C.1: Closure Benchmarking: Java Code

C.2 JavaScript

```javascript
/*
The function deferral measurements in node.js
*/

var deferredFunction = function ( numScopeVars, delay, scopeId ) {
  var scope = {};
  for ( var i = 0; i < numScopeVars; i++ ) {
    scope[ "id" + i ] = "12345678"; // 8 bytes per stored scope variable
  }
  setTimeout( function () {
    // If this is executed we didn't wait long enough
    console.log( JSON.stringify( scope, null, '  ' ) );
  }, delay );
}

var numOfFunctions,
    numOfScopeVars = process.argv[ 3 ];

numOfFunctions = Math.pow( 2,  process.argv[ 2 ] );

var time = process.hrtime();
for (var i = 0; i < numOfFunctions; i++) {
  deferredFunction( numOfScopeVars, 1000 * numOfFunctions, numOfFunctions + "(" + i + ")
};
var diff = process.hrtime( time );

var mem = process.memoryUsage();
// [...] benchmark system out
process.exit( 0 );
```

Listing C.2: Closure Benchmarking: JavaScript Code

Glossary

Mashup TODO. 3

Web of Things TODO. 1

Web Resource A Web Resource is anything in the Web which can be identified. Resources and their semantic properties are described through RDF in the Semantic Web. iii, 11, 12

Web Service Interface for communication between applications over a network. 3

World Wide Web TODO. 3

Acronyms

CEP Complex Event Processing. 8, 9

ECA Event-Condition-Action. 3, 8

IaaS Infrastructure as a service. 3

KR Knowledge Representation. 8

PaaS Platform as a service. 3

RPC Remote Procedure Call. 4

SaaS Software as a service. 3
SOA Service-Oriented Architecture. 3
SOAP Simple Object Access Protocol. 4

WSDL Web Service Description Language. 4

XML Extensible Markup Language. 4
XML-RPC XML - Remote Procedure Call. 4

UNIVERSITÄT BASEL

PHILOSOPHISCH-NATURWISSENSCHAFTLICHE FAKULTÄT

Declaration on Scientific Integrity
(including a Declaration on Plagiarism and Fraud)

Bachelor's / Master's Thesis *(Please cross out what does not apply)*

Title of Thesis *(Please print in capital letters)*:

____

____

____

First Name, Surname *(Please print in capital letters):* ________________________________

Matriculation No.: ________________________________

I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged.

I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

In addition to this declaration, I am submitting a separate agreement regarding the publication of or public access to this work.

☐ Yes ☐ No

Place, Date: ____________________________

Signature: ____________________________

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis .

UNI
BASEL