A RESTful architecture for integrating decomposable delayed services within the Web of Things

A. Ruppen*

Software Engineering Group,
University of Fribourg,
1700 Fribourg, Switzerland
E-mail: andreas.ruppen@unifr.ch
*Corresponding author

Prof. Dr. J. Pasquier

Software Engineering Group,
University of Fribourg,
1700 Fribourg, Switzerland
E-mail: jacques.pasquier@unifr.ch

PD Dr. T. Hürlimann

Decision Support Group,
University of Fribourg,
1700 Fribourg, Switzerland
E-mail: tony.huerlimann@unifr.ch

Abstract: The Web of Things research activities consist essentially in developing concepts, tools and systems for creating and operating global networks of devices associated with embedded resources - RFID tags, sensors, actuators and even complex computing facilities -, which are accessed by services. In that context, one of the most accepted standardization technique in order to seamlessly integrate this potentially huge set of heterogeneous services consists in RESTifying them. This operation is rather straightforward for "quickly computed" services such as Flickr photo service, Google Maps or Twitter. But what about cleanly integrating *delayed, possibly decomposable services*, such as a parcel delivery service computing first the partitions of parcels to various tours and then, for each tour, the optimal routing respecting additional constraints?
We claim that this question represents an important challenge if one desires to enrich the potentialities of the web of things. In order to best contribute to its solution, this paper first proposes a classification of services into five categories with a special emphasis on tackling the problematic of the delayed and decomposable ones. It also advocates the use of WebSockets as the standard callback mechanism. Secondly, it enriches the preceding theoretical discussion with three motivating examples. Finally, it presents a generic software architecture for RESTifying decomposable delayed services and validates it with a case study including a prototypal implementation.

Keywords: SOA, Web of Things, Internet of Things, RESTful architecture, Decomposable delayed Web-Services, WebSocket.

Reference to this paper should be made as follows: Ruppen, A., Pasquier, J. and Hürlimann, T. (2012) 'A RESTful architecture for integrating decomposable delayed services within the Web of Things', *Int. J. of Internet Protocol Technology*, Vol. 7, Nos. 1, pp.197–212.
Acknowledge: This paper is a revised and expanded version of a paper entitled 'A RESTful architecture for integrating decomposable delayed services within the Web of Things' presented at International Conference on Parallel and Distributed Systems (ICPADS), 2011 IEEE in Tainan, Taiwan

Biographical notes: Andreas Ruppen did his Master Degree in Computer Science at the Department of Informatics of the University of Fribourg (DIUF), Switzerland. He then worked for the Swiss Confederation on a project for the Swiss Federal Statistical Office. He is currently doing his PhD in Computer Science at the DIUF. His areas of interest are the Web of Things (WoT), SOA architectures, Software Design Patterns and Security.

Jacques Pasquier holds the software engineering chair at the Department of Informatics of the University of Fribourg, Switzerland, where he is currently Vice Rector responsible for the IT infrastructures and international relationships. Before joining the DIUF in 1987, he was a senior research associate at the T.J. Watson Research Center in Yorktown Heights, USA and a Swiss NSF scholarship student at the Mathematical Sciences Department of the Johns Hopkins University in Baltimore, USA.
Professor Pasquier current research interests include the design of Service Oriented Architectures (SOA) with a special emphasis on RESTful web services and mashups, the Web of Things (WoT), as well as the smart composition of context aware web services. His prior research activities focused on modeling complex reliability systems, software patterns and frameworks, hypermedia and new learning technologies, as well as distributed multi-agents systems. He is the author or co-author of three books and around thirty refereed papers.
T. Hürlimann is doctor in economics and has a venia legendi in Computer Science. He works as a research fellow in the realm of Operations Research and its applications and as a lecturer at the DIUF since 1991. He has developed the mathematical modeling language LPL. His main research interest includes Operations Research methodology, mathematical modeling, its industrial applications and implementations, as well as educational technology.

1 Introduction

Over the last decade, technology advances have allowed embedded devices to become cheaper and more sophisticated. This progress opened new perspectives. It gave birth to innovative ways of using such devices by attaching to them sensors and actuators. Adding networking capabilities was the next logical step, since it allowed companies to track shipment, production and storage of goods. Following that path, a real need came up to seamlessly integrate physical objects as well as their virtual representations into the Internet. Out of that, the Internet of Things emerged. Internet of Things means the Internet related to Things (Huang and Li, 2010). Another definition can be found in (Haller, 2010) and IOT-A (2012). By looking closer at the classification in (Haller, 2010), we see that the things are considered to be physical entities. Furthermore, they can be associated with devices which are either embedded or environmental. Finally, devices contain resources which can be accessed by services.

Key architectures for allowing seamless interactions between these resources are REST (Fielding and Taylor, 2002) and RESTful services (Richardson and Ruby, 2007). Indeed, RESTful interactions have several advantages over fully fledged WS-* web services (Pautasso et al., 2008; Guinard et al., 2009, 2010b; Drytkiewicz et al., 2004; Luckenbach et al., 2005; Guinard et al., 2010a). For example, since HTTP is used as application protocol and not simply as a transport one, a browser is sufficient for accessing and exploring resources. The latters are identified by URIs allowing to bookmark and share them. Besides, it allows the adoption of well established web technologies like authentication, security with https, integration with Javascript, etc.

Following that reasoning, services composed with things are usually RESTful (or at least RESTified) in order to seamlessly integrate them. This is for example the case for Flickr[1] photo service or Twitter[2], but what about cleanly integrating *decomposable delayed services*? In opposition to services like Google Maps[3], which are instantaneous and "atomic", decomposable delayed services might split themselves up into sub-services, each likely to be delayed. We claim that if we really want to bring an added value to the Web of Things (WoT), seamlessly integrating such services is important.

In this paper, we want to investigate how to seamlessly integrate these services into the Web of Things. The aim is to formulate a general architecture fulfilling the intended purpose. In order to best achieve this goal, the present paper is structured as follows: Chapter 2 proposes a classification of services into five categories with a special emphasis on tackling the problematic of the delayed and decomposable ones. In the last section of the chapter, a standard callback mechanism based on WebSockets is also discussed; Chapter 3 presents three motivating examples where delayed interactions, service decomposition and callbacks are needed and comments them within a RESTful perspective; Finally, Chapter 4 proposes a generic software architecture for RESTifying decomposable delayed services and Chapter 5 validates it with a case study including a prototypal implementation.

2 The Challenge of Decomposable Delayed Services

The Web of Things started with combining physical devices, sensors and actuators in a standardized manner. This had the advantage that devices from different sources could easily be combined without the hassle of

learning each time about the vocabulary used by the service. The WoT, however, is not limited to physical devices. Indeed, bringing delayed, possibly decomposable services, into the WoT world allows for the creation of much more sophisticated scenarios as we shall see.

Integrating services such as *Flickr* or *Twitter* into the Web of Things is rather straight forward. It is just a matter of one request. This, however, does not apply for *decomposable delayed services*. Their integration into the Web of Things raises new challenges, which must be solved. The need for integrating delayed services into the Web of Things is not new. In the following, we give an overview of the problematic of delayed interactions and discuss some standard approaches to solve the problem.

Integrating services into the Web of Things works most of the time out of the box. Drawing coordinates obtained from Google Maps for example, is just a matter of sending a `GET` request with some URL encoded values. The result is delivered immediately. However, the three examples of Chapter 3 will show that services can sometimes be quite complex and might require a long delay to execute and to deliver the desired output. For our purposes, let us roughly classify services into five categories, each of them having its properties and special requirements regarding the architecture to choose in order to integrate them.

2.1 Short Living Services

The first category includes the *short living* services. Even though some computation is eventually done upon requesting the service, the result is almost instantaneous. An example of such a service is the routing service of Google Maps. Each time a route is requested, the service launches a computation and returns the computed routing between the submitted locations. This is so quick that it does not make sense to cache the result for later consultation. In fact, a user cannot distinguish if the answer from the server is actually computed or if he has accessed a static resource. Such services integrate seamlessly into the Web of Things. Consuming them is just a matter of a *GET* request.

2.2 Real-Time Services

The second category is composed of the *real-time* services. Upon requesting such a resource, the user gets immediately its representation. Over time, however, the resource might change and the user would like to be informed about updating events. Since these happen constantly, it is not necessary to worry about problems like connection timeouts or users quitting the interface. An example for such a service is the Twitter timeline. It can be accessed in a RESTful manner. This resource, however, changes over time as new tweets arrive. Therefore, its representation changes also over time. A popular approach to offer such services consists in using web-hooks: upon requesting a resource, the user provides a URI under which the service can contact him and provide him with updates. While this solution proposes a clean design, it has the drawback that it most certainly will not work in corporate environments. Security policies will most of the time forbid opening ports for incoming traffic and therefore block the updates from the service. Another well established approach is *Comet* (Comet Framework, 2012). The Comet framework uses either HTTP streaming or long polling in order to fetch the updates from the server. Unfortunately, Comet is not supported at the same level in all browsers, which makes it a big and complicated framework. The upcoming HTML 5 also brings a set of new technologies to the browser, the *WebSockets API* (WebSocket API, 2012) standard being one of the most promising. A WebSocket opens a new TCP/IP connection in the browser. The connection is full-duplex, which means that communication is possible in both directions at the same time. Besides, the connection is Firewall and NAT proof, which makes it a real alternative to web-hooks. Upon requesting a resource, a new WebSocket is created. Later notification about updates of the resources are sent over this WebSocket together with the new representation of the resource. Actually, the main drawback of this solution is that the WebSocket API is currently only a draft version from *W3C*, with the consequence that its support in modern browsers varies.

For these two categories, even if the service executes some computation on the data passed along with the URL, the result is computed at each request. This is smart insofar as the time needed for the computation is barely measurable. The service is REST compliant in the way that the URI to the result can be bookmarked and viewed again. For such instantaneous interactions, it would not make sense to store the results somewhere in the cache. It is much more practical to compute them each time. In the case of delayed services, however, this is clearly not the best approach.

Short living and *real-time* services are the base of the Web of Things. Querying a sensors for its current state can either be seen as a *short living* service if the user is only interested in its value once or it can bee seen as a *real-time* service if the user is more interested in the value changes from the server. The frontier between these two categories is fuzzy and sometimes only the concrete use-case defines in which of the two categories an application actually fits better.

2.3 Delayed Services

The third category are the *delayed* services. When the user requests a resource, the service is unable to send back a representation of the latter in a reasonable time. An example for such a service is the planning of Round-Robin Tournaments with complicated constraints like minimizing breaks (Briskorn, 2008). Finding a solution is time consuming. In fact the computation launched by requesting the resource is so time consuming that no user would really wait for the result to be ready. Besides

that, such long delayed services might lead to dropped connections. Yet, a dropped connection does not stop the computation on the server; thus, CPU cycles are wasted. Furthermore, the user will see the dropped connection and most likely will relaunch the request. This multiplies the server charge needlessly, which may lead to a break down of the system all together. One possible solution would be to use longer connection timeouts on HTTP connections. However, for usability reasons, this is a bad idea: the user sends a request to the service and thus expects a result. But, if the process takes too much time, the user tends to think that there is a connection problem and she aborts. Another point is that systems using long HTTP connection timeouts scale very badly. This is surely not in the philosophy of the Web of Thing where thousands of devices are supposed to interact.

As for the second category, web-hooks are a common approach to solve these problems. However, if we cannot expect the user to wait for the final result, we cannot expect that the provided web-hook will be available when the computation finishes. Another approach is proposed by Richardson et al. (Richardson and Ruby, 2007). Instead of requesting the resource with a `GET` statement, a *new task* is created with a `POST` statement. The latter shows the progress of the ongoing computation and, ultimately, its result when finished. With this solution, the server should always respond to a request, either with the computed answer or, if the computation takes too much time with a *202 Accepted*, plus some URI where the user can check the progress and, if available, the result. We do consider Richardson's solution as the most rightful one at the architectural level and we will elaborate on it in Chapter 4. An advantage of this proposition is that it is always possible, as an add-on, to use either the Comet framework, polling or WebSockets in order to track the progress of the computation.

2.4 Decomposable delayed services

The fourth category is composed of *decomposable delayed services*. Upon being requested, such a service does not respond with an "atomic" answer, but rather split itself up into sub-services, thus returning a whole hierarchy of answers. Furthermore, as we shall see in the examples of Chapter 3, each sub-service might have its own delay. This fourth category is the one on which the rest of this paper focuses. It represents clearly a generalization of the third one, for which Richardson and Ruby's solution (Richardson and Ruby, 2007) must be further refined. Today, there exists no standard approach to solve such problems. Our solution will combine several known and well established techniques from the WoT world in order to provide a clean integration of such services into the Web of Things.

2.5 Business process services

The fifth category are the *business process services*. Services in this category don't fit in one of the other categories. Their main property is that a business process is launched when they are invoked. This can be just a small computation or a heavy use-case with many dependencies. Business process have to handle properly cases in which the computation cannot be completed (if for example some input are not available). An example of such a service would be the booking of a trip abroad. A trip is in general composed of the flight plus a hotel reservation. The trip can only be booked if a flight is found and booked and if a hotel is found and successfully booked. Since flight tickets and hotel reservation systems are updated frequently, it might happen that a user selects a flight and a hotel, successfully books the flight but when arriving at the stage of booking the hotel, no more rooms are available. Such cases needs special care since the system should not be left in an inconsistent state. Another example is the electronic shelf labels in a shopping center (Magerkurth et al., 2011).

In a certain way, this last category represents the ultimate generalization of the preceding ones. As we shall see in the next chapter, a lot can be achieved with single delays and decomposition, but sometimes complex business services needs to be interrupted until a given event wakes them up and helps them pick-up the next action or they need to be rolled back up to a given point. How to model, architect and implement such services is out of the scope of this paper and an active ongoing research topic (Sperner et al., 2011). The currently established tools for modeling such business process lack of support for the Internet of Things related entities like sensors and actuators. First efforts have been done by adding some basic IoT concepts to BPEL but much remains to be done (Meyer et al., 2011).

2.6 Callback mechanisms

For some of the categories above it would be interesting to have a sort of callback when the service reaches critical points or when it is done and the results are ready. There exist several approaches to solve this problem. One could imagine an email integration into the process. At each predefined step, an email is sent to some address (which might be defined as input when requesting the service). Similar approaches consist of protocols like XMPP (XMPP Protocol, 2012; RFC 6120, 2012; RFC 6121, 2012; RFC 6122, 2012). Even if these protocols works well and are widely used for other means they present the disadvantage of being out of the scope of the WoT. Web-hooks are also a popular approach to inform the user about events. As for the email callback, the user has to provide some information about the hook where the server can join her later. Basically, a web-hook is a client side resource which is made available to the server. Even if web-hooks are cleanly integrated into the Web of Things they have the drawback of being banned from corporate environments. In fact, most firewall will block incoming connections on random port numbers making the web-hook use-less.

Finally, together with HTML 5 a new standard for full duplex communication in browsers has been recently established: the WebSocket protocol and API (WebSocket API, 2012). WebSockets allow asynchronous communications in a browser. They are transparent to firewalls and proxies, making them a good alternative for the solutions presented above. A WebSocket connection is always initiated by the client. Thus, she can decide when she wants to listen to the notifications generated on the server. Furthermore, it is possible to re-connect to the notifications at any time. This makes WebSockets a flexible mechanism for providing callbacks and feedback from services. They are really well in sync with the Web of Things philosophy. They will be further discussed in Section 5.2.

3 Three Motivating examples

3.1 Smart Shopping List Service

The introduction of the first iPhone and iPod Touch by Apple in 2007 changed the way we solve everyday problems. Formerly, shopping lists were written down on small pieces of paper with no apparent order of the items. Nowadays, there is a shift to manage such shopping lists on smart phones. There are plenty of applications developed for this task. They have several advantages over the paper version. Besides the fact that the users carry their smart phones always with them, they offer functions like barcode scanning for an easy setup of a new shopping list and some propose options to assign given products on the list to local stores. Yet, they present the same disadvantage as the paper version: there is no "smart sorting" of the items to purchase (e.g. ordering by stores and within a store by shelves, with various routing optimization and so on).

On the other hand, it is possible to build services which can assign products to local stores, either automatically or by the user's preferences and then compute an optimal order to visit these stores. Furthermore, most shopping centers are layed out following some psychological guidelines (e.g. fresh fruits and vegetables right at the entry). The services could take into account this knowledge and propose an order of the items for each store to visit.

We could imagine a smart shopping list application where the user can add products either manually by providing a name or by scanning an already purchased item. Each product on the list can be assigned to a shop. If the user assigns a product to a store, this item has to be purchased on that store. Once the user has entered all the desired products, the list should be saved and sorted in a logical manner.

We can define the following hierarchy for locating a product. Each product is available in at least one store. Such a store is made of shelves presenting the different products. A store is found either at a given address or in a bigger shopping mall. A shopping mall has a physical address. Based on this hierarchy it is possible to find out a smart order of the items to purchase.

Such a system first has to assign each product to a store. This is done either by respecting the user's preferences or by smart-selecting a shop selling the product. This smart selection has to take into account that the number of different shops to visit should be kept low. Once each product is assigned to a store, the best routing between the shopping malls to visit is computed. At the same time, for each shopping mall, there exists a best order to visit all the shops. Furthermore, each shop is layed out according to known principles. This allows to roughly order the list of products for each store. All these tasks can be done in parallel.

Therefore, a shopping application could integrate such a "routing service". When the user has finished adding products to its shopping list, the service is called with the list of products as parameters. The service would then use the necessary computing time to return a "smart list" sorted by shopping malls, stores, shelves and products. The client can now go shopping. Arriving in one store he can grab the ordered list for this store and do the shopping.

Note that a smart shopping list service has to solve several sub-problems in order to solve the overall one of smart ordering the list of items to purchase. Most of these sub-problems will only have a short delay and the overall delay for solving the problem is not the primary matter in this use-case. Yet it illustrates well the necessity to decompose the primary task into sub-tasks in such a way that it will be very comfortable for the design of mashup applications to have a different URI to each resource (e.g. one for accessing the whole shopping list, several sub-resources for each shopping mall and several sub-sub-resources to the items to purchase in each store).

3.2 Alert system for Firefighters

Let us imagine a system for alerting firefighters of an incident. For that purpose we will base our description of the system on the structure and roles of the British fire service. A fire department is organized as a hierarchical structure. For each area there is an *Area Manager* responsible for several *Groups*. Each *Group* is made of a given number of *Stations*, each having associated some firefighters and vehicles (Firefighters, 2012; Firefighter Roles, 2012; British Firefighters, 2012). Figure 1 shows these hierarchical dependencies. When an incident is declared a specific number of firefighters and vehicles have to be sent to the incident and the *Area Manager* decides which *Group* will handle it. The *Group Manager* will then decide upon the *Station* having enough capacity to handle the incident and finally, firefighters from one station are sent out. Several factors influence the choice of one *Group / Station* or another, among them, the distance to the incident and the manpower of a station. The alert is only treated as soon as one station successfully committed.

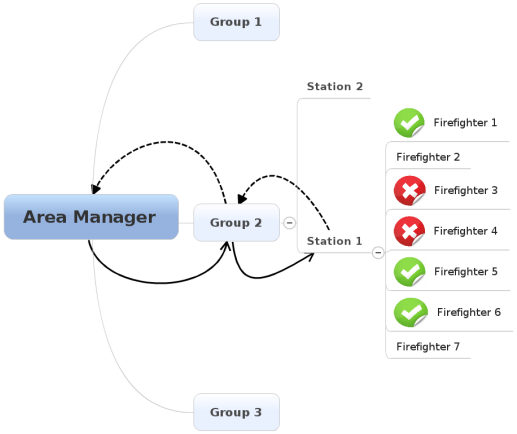

Figure 1 Hierarchy of a Fire Department

This scenario embeds seamlessly into the Web of Things. The vehicles and the firefighters can be seen as *Things*. Vehicles can be tracked by GPS in order to know if they are currently in use or not. The same applies for firefighters. With a small application on their smart-phones the system knows whether a firefighter is already handling an incident, on holiday or available. In countries where there are no professional firefighters the biggest challenge is to find enough man-power to respond to an incident. Today this is done by pager or phone calls or sometimes by SMS. In a world seamlessly integrated into the Web of Things, this process could be greatly simplified. As soon as an alert gets recorded it has to be treated by the hierarchy described above: The area manager creates a new alert in the system. The system then transmits the alert to a suitable group, which in turn selects a station and by that the attached firefighters. The system is smart enough to discard groups and stations which have no capacity to handle this alert. By having a persisted alert in the system, the area manager or a group manager can always check in which state an alert is. On the last layer, each firefighter gets the alert. The alert message that the firefighters get consists of a link to the alert resource. By clicking on that link a new WebSocket connection is opened to the alert on a *Station* level. Thus, each firefighter can see if there are still more firefighters needed. If this is the case, he can accept the alert and by that increase the number of committed firefighters for that alert. Each firefighter connected to the alert will get updates about other commitments in real-time. As soon as enough firefighters have committed to the incident, the alert is marked as treated by a station. This in turn closes the alert on a group level and by that closes the alert for the area manager. The area manager can come back later and check which group finally has accepted the alert. The system has to be smart enough for handling situations where a suitable station is found but not enough firefighters committed in a reasonable time. In such cases, the alert has to be escalated to the next available station or group. As for the firefighters a station manager can open a WebSocket connection to the alert resource in order get informed about who has committed.

Therefore, the process of finding firefighters for handling an alert decomposes itself into several sub-tasks: finding a group handling the alert, finding a station and enough firefighters to accept the alert. In addition, the handling is done asynchronously as it might take some time to find the right amount of firefighters.

This use-case shows another interesting problem where decomposable and delayed services are needed. Since fire departments have to handle incidents as quickly as possible, the delay is in general not very big. Yet, it is interesting to see it as a delayed service. When an incident arrives at the area manager he has to take care of it. However, he cannot always wait for the result of pushing the alert down to a station as other incidents might happen. Thus having a sort of delayed system where he can check later the outcome of that alert is necessary. Besides, decomposing the problem is necessary in order to give each participant direct access to the relevant information. This might be the accept/deny part for a firefighter or the overview of how many firefighters did accept the alert for a station manager. Furthermore, this mechanism allows for a hierarchical escalation of the alert to ensure that at least one station will take care of it.

3.3 Parcel Delivery Central

A parcel delivery central receives each day a large number of parcels, which have to be delivered to addresses the next day. Upon arriving at the postal office, each parcel is scanned (for example by reading its RFID tag or its bar-code). Through this scanning the system knows which parcel has to be delivered to which address. The parcel is then added to the current distribution list. In the evening when the office closes, all parcels have to be assigned to different tours. Moreover, each parcel has a given priority, which gives an additional constraint on the delivery time.

This is an instance of the well known problem in operations research called *Vehicle Routing Problem with Time-Windows (VRP-TW)* (Golden et al., 2008; Paolo and Vigo, 2002). It consists of finding an optimal set of routes to be performed by a fleet of vehicles to serve a given set of customers. The problem can be described as follows: Given n locations (the customers and the parcel delivery central) numbered from 1 to n, we want to deliver goods to each customer in a given window-time using k vehicles, all starting at the central (location 1) and returning to the central using the shortest distances possible that the vehicles have to travel. This problem is NP-complete and it is very difficult even for small instances to find an optimal solution. The problem can be solved in two steps:

1. partitioning all the customers 2 to n into k subgroups and
2. finding the shortest tour for each subgroup.

Immediately after the first step is finished and thus the partitioning known, the parcels can be distributed to the different trucks. Each truck has an assigned tour and driver. The next morning when the drivers are arriving, each one can check its assigned route and deliver the parcels according to the computed routing and their priority in an optimal route.

This scenario clearly shows the interest and need for decomposable delayed services. The scenario describes how the problem has to be decomposed into sub-problems in order to solve the first instance of the problem. Additionally each sub-problem has its own delay which varies from one sub-problem to another. In the worst case scenario the overall delay for solving the problem is the sum of all sub-problem delays.

4 Generic RESTful Architecture

Chapter 3 introduced three motivating examples of decomposable possibly delayed services. Even if each service provides its own specific features, there are important similarities, which can be abstracted in order to obtain a generic architecture for decomposable delayed services. This section explains our architecture in two steps: first, we define the structure of a *task* and second, we show how to seamlessly integrate a RESTful façade to an arbitrary delayed service.

4.1 Basic RESTification

The idea of using tasks on a server representing the ongoing and finished jobs is not new. A first approach to this problem has been presented by Richardson et al. (Richardson and Ruby, 2007).

To our point of view, it is a good reasoning, but not enough. What happens for example, if the created task decomposes itself into several sub-tasks (see the Parcel Delivery Center example in Section 3.3 and its prototypal implementation in Chapter 5)? Indeed, resources that appear and disappear are very much in the spirit of the Web of Things where many things and processes interact simultaneously. As stated in (Richardson and Ruby, 2007), there are not that many possibilities to bring asynchronous services into the world of RESTful services. Each request to service x creates a task, which is added to the task queue associated with the service. This queue and its tasks are the only resources bound to the service x. As Richardson et. al raise the task is not a state which is transferred to the server (and thus would violate the concept of RESTful interaction), it is a newly created resource with which a client can interact. The associated URI scheme is therefore quite simple: *http://.../servicex/tasks/* is the list of all tasks on service x and *http://.../servicex/tasks/id_y/* points to a given task y. This definition can be extended recursively. As stated by RESTful principles, the representation of a task should contain links to related resources, in this case the associated sub-tasks. Thus, *http://.../servicex-/tasks/id_y/id_z/* points to one sub-task z of task y. This recursive definition can be drilled down to as many layers as needed. Figure 2 gives a conceptual overview of this hierarchy of tasks. Let us have a closer look at the shopping list example of Section 3.1. The URL *http://.../shopping_list/tasks/* would return a list containing all shopping lists created on the server. Whereas *http://.../shopping_list/tasks/jp1/* would return one particular shopping list containing, among others, a list of links to the shopping lists for each store to visit. Finally *http://.../shopping_list/tasks/jp1/store1* would return, among others, the list of products to purchase in *store1*.

A deeper analysis of the examples of Chapter 3 shows that each presented service makes use of tasks and task lists. Besides, all tasks share a common set of properties. Therefore, we propose that a task is always defined by at least the following set of attributes:

- A unique identifier is needed to identify a task. This identifier is also part of the URI pointing to this task. It is the only mandatory attribute.
- Additionally, the user Id is part of these attributes. It ensures a clear separation of users and rights.
- Furthermore, the results are also part of the list of attributes. They may not yet be available and thus be empty.
- Just as important is the input. It describes the actual task which has to be performed by the system.
- Once created, a task is executed by the service and, after some period, the result is available. Therefore, a task needs a status indicating whether it is still running or not.
- Finally the start-, end-time, and a short error message are not directly part of the task. However, it is interesting to have access to this information.
- Since a task can have sub-tasks, they have to be linked against their father. Therefore, a task should contain a list of pointers to its related sub-tasks.

Most of these attributes are provided by the server. For example, the user Id should, for security considerations, be filled-in by the server. Similar reasons apply for the other attributes, except the problem description which is the only one provided by the user. As showed by Figure 2, tasks are defined recursively. A task can have several sub-tasks. This relation has to be translated into the system.

4.2 Additional semantic

In order to fully RESTify a delayed service, we first have to ensure that it will only be requested with a `POST` request. As usual, a `POST` request creates a new resource

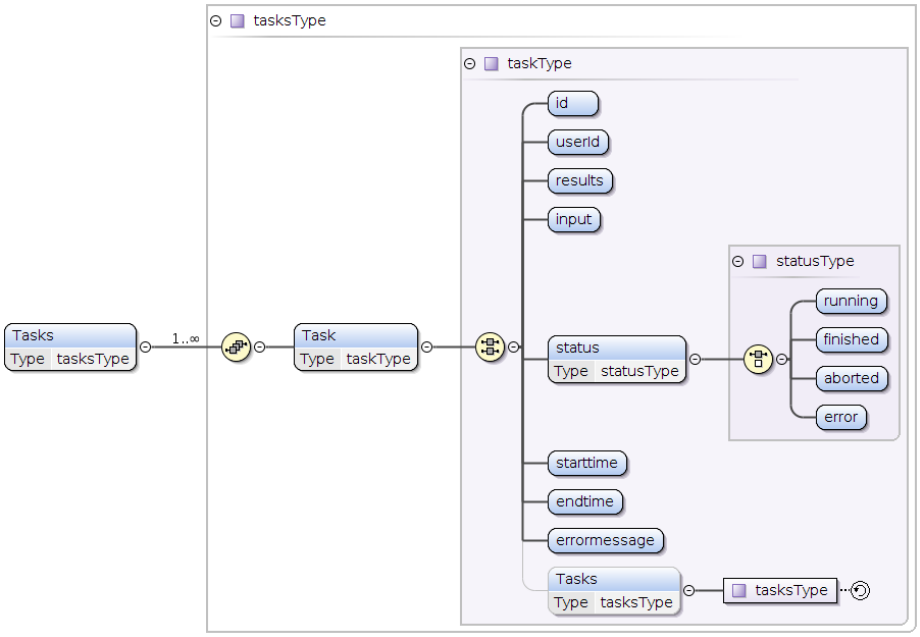

Figure 2 Conceptual overview of the tasks hierarchy

on the service. It is important to use the `POST` statement since it is not known in advance under which URI the new resource will be created. This resource represents the actual task that the service has to solve. Since the server replies with a "202 Accepted", the resource is created immediately and can, from this point on, be requested to know the state of the task. Since the resource describes an ongoing task on the service, its representation changes over time. The `POST` request should allow at least two content types:

1. XML/JSON for simple tasks and
2. multi-part form data for file upload.

The multi-part creation comes handy when the description of the computation is complicated or when the interaction with the service is file based (see Section 5.2 for more details). This semantic is fully compatible with RESTful principles.

While the `POST` request allows the creation of a task, its modification requires the `PUT` request. One possible modification of a running task that should absolutely be implemented is its abortion. There are situations where this is mandatory. This is for example the case when the service is blocked at some point of the computation. This can have many reasons: waiting for not available input, infinite loop etc. By sending a `PUT` request we can abort such a computation. Abortion of tasks is also interesting when launching problems which do an iterative refinement of the solution. Maybe at some time, the solution is good enough for the present situation and should stop to free the occupied computing facilities for other tasks. The third operation, `DELETE` removes a task from the service. When launching this request, the service should ensure that if the task is still running, it gets first aborted and then deleted. *Both, the `PUT` and the `DELETE` request should send back a representation of the task.*

Finally, the `GET` request returns either a list of tasks or a task and its list of sub-tasks. In the first case, it returns just a list of links pointing to individual tasks. In the second case, it returns both the attributes of the task and a list of links to its sub-tasks. As used in RESTful services, the `GET` method should be able to deliver different representations of its resources. Among them, JSON/XML and HTML should be available. Besides that, the `GET` method should allow filtering relative to most of these properties. A common use-case for this is searching for errored or unfinished tasks. Filtering for these attributes allows the creation of Atom (Gregorio and de Hóra, 2007) feeds to which users can subscribe to get, for example, all unfinished tasks or all tasks started before a given time (Wilde and Marinos, 2009). Since the introduction of HTML 5, a new approach for delivering updates is available: the WebSocket API. WebSockets allow a bidirectional communication between a client and a server. In our use-case of decomposable-delayed services, the bidirectional part is not the most important aspect of WebSockets. Their main advantage in this context is their ability to "push" messages from the server to the client. Hence, the `GET` method is the entry point for the WebSocket connection. Instead of polling for updates on a given resource, a user can open a WebSocket on this resources and by that, getting all updates on the resource. This can be handy for providing log messages to the user in real-time. WebSockets are not just a new content-type to handle. They will later allow the creation of mashup applications which will feel like native ones.

The discussion above shows that a task can be aborted, running or finished. Besides that, it is possible

that a task produces an error and stops. This can have several reasons like waiting for input which is not available etc. These considerations lead us to affirm that the status attribute must accept at least one of the four values: running, finished, aborted, or error as shown in Figure 2. In the case of an error, the error message attribute should provide some debugging information. The start- and end-time are not directly related to the task, yet, they can be used for example, to detect tasks which run for quite a long time and are therefore, good candidates for infinite looping tasks. Another use-case is the billing of used service time, which can be calculated based on these attributes. Depending on the situation, it is quite possible that additional properties are needed to describe the current status of task running on a service. Such attributes are situation specific and hence, out of the scope of this architecture.

4.3 Software Design Considerations

When building a RESTful delayed service, the decision of RESTifying the current service or building a new one from scratch has to be taken. Both approaches present their advantages and disadvantages, but both have in common that they provide a universal API to access delayed services in a RESTful manner. In most cases; however, the service already exists. This is especially the case for complex services where the business logic is complicated and developed for years. It can be seen as a black box to which a RESTful façade is created. Figure 3 shows a service with a WS- (or RPC) interface. The façade is built following our generic approach. It implements the WS-, respectively the RPC client-interface on the server side and offers a RESTful interface to its clients. Our approach makes it easy to create such façades. All the business logic remains in the original service. Therefore, the façade only acts as a mediator between a client and the existing service. This method also applies in situations where the black box service offers some proprietary interface only.

Either way, the architecture stays the same. On the client side the façade offers a REST interface respecting the principles presented in Section 4.1. On the server side, either the façade implements the required business logic or it implements a client (WS-*, RPC, POJO etc.) for an existing service which actually solves the problem.

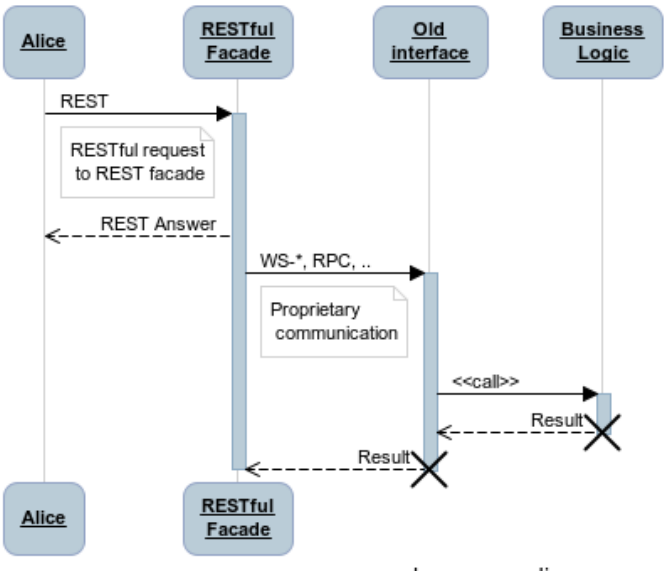

Figure 3 RESTifying a service

5 Case Study

5.1 Business Aspects

We now come to a concrete case study motivating the value of decomposable delayed services. Let us consider again the Vehicle Routing Problem with Time-Windows (VRP-TW) of Section 3.3. As already stated, the problem can be solved in two steps:

1. partitioning all the customers 2 to n into k subgroups and assign each subgroup to a vehicle and
2. finding the shortest tour for each subgroup starting at location 1 and returning to location 1 by satisfying the given time-windows for each costumer.

This problem has many practical applications in distribution and logistics. Suppose our central has to deliver letters and packets to 199 customers in a given region during a given day. Starting in the morning at 5h00, each of the four postmen (using a vehicle each) visits a subset of all customers. Suppose furthermore, that each customer is in a group of "express", "normal" or "slow" delivery, where "express delivery" means that the customer must be visited not later than 9h25, "normal delivery" means not later than 11h05, and "slow delivery" can be delivered the whole day till 16h00. Hence, we have given a set of customers including the central, defined mathematically as $I = \{1 \dots n\}$ with $n = 200$, where $i = 1$ is the parcel delivery central (the depot) and $i = 2 \dots n$ are the customers and we have $k = 4$ vehicles. Finally, we have the distances between the customers i and j given as a matrix $c_{i,j}$ with $i, j \in I$ (that is, a 200×200 data matrix) and a vector $p_i \in \{1, 2, 3\}$ with $i \in I$, saying what type of delivery the customer i asked for (1 = "express", 2 = "normal", 3 = "slow").

To solve this problem, we run in a first step a service that takes the data ($i \in I$, $c_{i,j}$, p_i and the number k) and distributes the customers into $k = 4$ groups. The result of this service is a vector $q_i \in \{1, 2, 3, 4\}$ with $i \in I$, defining the assignment of each customer i to a postman (vehicle), where $q_i = 1, 2, 3, 4$ depending on whether the customer i is assigned to postman $1, 2, 3$ or 4. In a second step, we run k services (in parallel) that take the data ($c_{i,j}$, p_i, k and q_i) and generates the *optimal* tour for each postman.

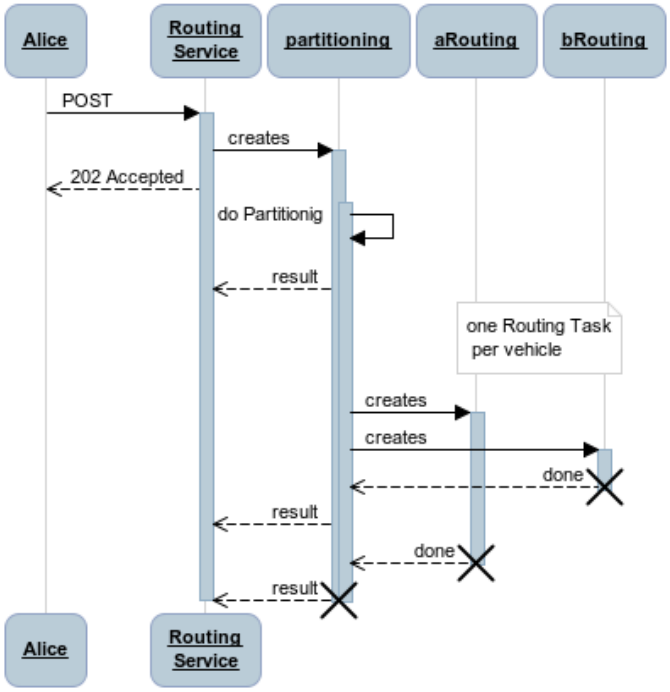

Figure 4 Creating a new Task on the VRP-TW service

This scenario embeds seamlessly into the Web of Things. Each parcel has a tag (for example RFID) containing among others, its priority level and the destination address which is read on entering the central. It allows, with the help of Google Maps (another service), the construction of the distance matrix $c_{i,j}$ needed later. The priority of each parcel and the number of available vehicles are sent together with the distance matrix to some "routing service". This service will, first compute the partitioning of the customer on the k vehicles and then define the best routing for each of them. Figure 4 gives a rough overview of the sequencing of such a service. As described above, to respect the idea of RESTful services, the routing service creates an URI to the partitioning task, as well as k sub-URIs to each of its routing tasks. By simply sending these URIs to the postmen, the latters can consult the status of these tasks and finally their routes on any web-enabled device. It would even be possible to mash-up this information with Google Maps and GPS location to create a turn-by-turn application always indicating how to reach the next client. Moreover, such an application could reflect the delivery status of each parcel.

Several lessons can be retained from the discussion of the case study above. There are a certain number of things. These are the parcels and their tags but also the vehicles. Apart from the things, there are synchronous services like the tag reading, the construction of a distance matrix and GPS tracking of the vehicles. Besides these synchronous services, there is also an asynchronous one which coordinates the whole, the routing service. However, everything is identified by an URI and all interactions are RESTful. Therefore, they present a seamless way to be integrated, thus creating a seamless Web of Things and Services.

5.2 Implementation Aspects

To validate our generic architecture of Chapter 4, we have implemented the Routing Service of Section 5.1 as a RESTful decomposable delayed service. Currently there is a program called LPL (Hürlimann; Hurlimann, 2012) for solving the VRPTW, but it is only available as a Windows DLL. Thus, it cannot be used directly from a mashup. Optimization problems, as in the case study, are described by models and their associated data on which the computation will be done. We applied to this DLL our generic solution to RESTify it. The model is already provided as LPL binary. Thus, the only input needed are the data on which the computation will be executed. The choice between starting over or implementing a façade was easy here. The solver has been developed for years and is very stable in its current form. However, the solver has no web interface. Thus, we created a RESTful façade for it.

All the methods mentioned in Chapter 4 were implemented. The LPL binary accepts two sorts of input, plain text and a pre-compiled binary format. Thus, the `POST` method accepts either XML/JSON input, or multipart form-data. The first content-type sends plain LPL commands embedded in the XML, whereas the latter uploads a pre-compiled binary in LPL format to the service. Both were implemented, but currently only the first one is used. Upon posting a new task, the service fills out the missing fields like the start-time and the user ID and persists the task into a database. Afterward, the LPL solver is launched with this data

```
<Task uri="http://localhost:9090/SlowLPLServer/resources/tasks/617506558" status="finished">
  <id>617506558</id>
  <userid>rua</userid>
  <result>
    <fileResult>http://localhost:9090/SlowLPLServer/lpltmp/617506558/617506558.nom</fileResult>
    <imageResult>http://localhost:9090/SlowLPLServer/lpltmp/617506558/617506558.jpg</imageResult>
    <logResult>Current shell is...SUBTASK ended 5</logResult>
  </result>
  <input>http://localhost:9090/SlowLPLServer/lpltmp/617506558/617506558.lpl</input>
  <startdate>1325607398832</startdate>
  <enddate>1325607418966</enddate>
  <usedSolverTime>15</usedSolverTime>
  <problemName>Test 2</problemName>
  <numberOfSubTasks>4</numberOfSubTasks>
  <subtasks>
    <ShortTask uri="http://localhost:9090/SlowLPLServer/resources/tasks/617506558/1/" status="finished" />
    <ShortTask uri="http://localhost:9090/SlowLPLServer/resources/tasks/617506558/2/" status="running" />
    <ShortTask uri="http://localhost:9090/SlowLPLServer/resources/tasks/617506558/3/" status="finished" />
    <ShortTask uri="http://localhost:9090/SlowLPLServer/resources/tasks/617506558/4/" status="running" />
  </subtasks>
</Task>
```

Listing 1 XML representation of a Task

set as input. This is the first part of the computation where the partitioning of the parcels is computed. As soon as this first step - partitioning into 4 tours - has finished, the task attributes are updated, reflecting the new state as well as the result. Figure 5 shows a graphical representation of a partitioning while Figure 6 displays the tour associated with the second vehicle. The attentive reader will notice that since each parcel has an assigned priority, it is not enough to partition the space and return the shortest path for each turn.

The partitioning computation took 10 minutes on a standard dual-core machine. With this result as input, the second part of the computation is started, computing for each of the k vehicles the best routing. This is done in parallel for each of the k vehicles. To respect the RESTful principles, a new task is created for each of the vehicles. They get the same user ID as their parent task. These tasks will later be consulted by the postmen to get the details of their tour. In our example, where n is 200 and k is 4 the computation of a solution took about 20 minutes, pushing the overall running time to at least 30 minutes. This amount shows that a synchronous solution would not be apt to integrate this service seamlessly. Accordingly, the parcel delivery office can accept parcels until 3.30 AM. Passed that deadline, the service needs to be started to know the tours in the morning when the postmen arrive.

The `GET` method returns either the current status of the task, including a set of links to the computed result if already available, or a list of tasks, depending on the URI on which the request is executed. This request, as RESTful principles propose, delivers either a fully fledged HTML page or XML/JSON data which can then be used to be integrated into a mashup on the postmen smart-phones. Listing 1 shows the XML representation of such a task. The listing displays a finished task which has four sub-tasks, two of them are still running. All the attributes summarized by Figure 2 are present and correctly filled in. Furthermore the chosen URI schema is

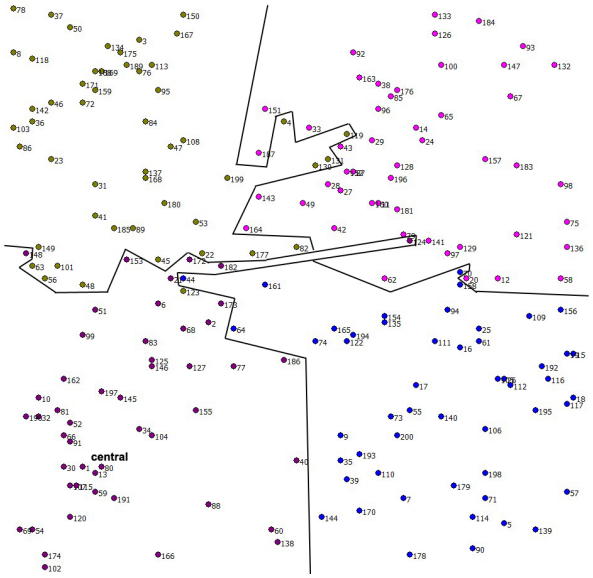

Figure 5 Partitioning of the parcels

similar to the one presented in Section 4.1. The input is no longer available as plain text but was transformed into a file read by the solver. This is mainly for two reasons: 1. on the server side, the communication with the LPL solver is file based, thus, even if the task is created from

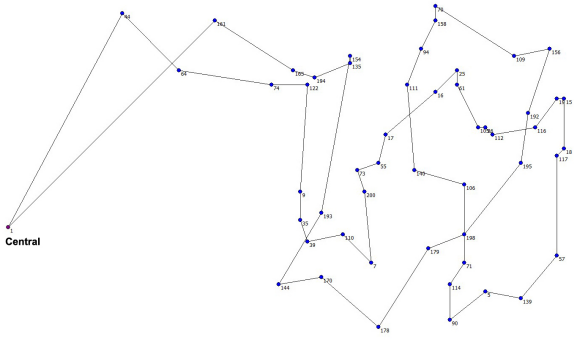

Figure 6 Computed tour for vehicle 2

its XML representation, an input file for the LPL binary will be created; 2. returning always a link to the input file allows for an easier integration of the service into future mashup applications. As stated before, the result is a list of links pointing to the individual parts of the solution. In fact, besides the information needed for the delivery (`fileResult` and `imageResult`), the LPL solver has also computed information needed for quality management of the service (`logResult`). The complexity of the result attribute however, may vary from one service to another. The attentive reader will also notice the links towards the four sub-tasks (finished or still running).

Additionally the `GET` method is also the entry point for a WebSocket connection. This is achieved by using the media type *text/plain* for WebSocket connections. If a browser requests a plain text representation of a task, the server will initiate the WebSocket handshake with the client. This media type is only available on a given task but could easily be expanded to task-lists or subtasks. After a successful handshake, all the log messages generated by the solver are transmitted to the client over the WebSocket thus, allowing him to follow closely the process.

Besides, it is possible to add filter queries to the request: hence, only a partial task list is returned. Again, this information is not helpful for the postmen but rather to the office launching the tasks. Such queries make it possible to find quickly erroneous computations and react accordingly.

The `PUT` and `DELETE` requests modify an already-existing task. Both methods return the same information as the `GET`. However, the `PUT` method is used to abort a currently running task whereas the `DELETE` aborts and removes a task, independently from its current running status. Since a task can have sub-tasks, resolving sub-problems, a `DELETE` also removes all sub-tasks. This has to be done to ensure integrity of the underlying data structure. Both methods have to take care that the launched process (i.e. the solver) also gets properly stopped and resources cleaned-up and freed to give other running task more computing facilities to run.

The implemented façade contains only the code interacting with the actual LPL solver and generating the recursive tasks. No business logic is implemented in the façade. Furthermore, the attributes discussed in Section 4.1 are sufficient to describe the system. They add enough information to ensure that the service is working, but not more than needed. The attentive reader may notice that there are a few more attributes on Listing 1. They are mainly intended for quality control and billing. The prototypical implementation proved the validity of the presented generic solution. The implementation provides now a standard and easy way to use our API to access the LPL solver. Since REST is a stateless protocol, it is needless to wait for the task to finish. Instead, it can be launched and the status can be checked later on.

6 Conclusion

We identified the challenging problem of integrating decomposable delayed services within typical Web of Things situations. We were able to provide a generic (fully RESTful, using established standards) architecture to its solution and we validated the whole with a prototype. We also showed with our façade approach that such a service can be RESTified even if you are not its owner.

Besides the prototypal implementation of the parcel delivery service we have several other ongoing projects in this domain to prove the viability of our approach. A system similar to the alert system for firefighters presented in Section 3.2 is currently developed together with the University of Applied Science of Western Switzerland and datamed SA[4] as part of a CTI[5] project. The focus of the project is to build an e-Health system providing the medical staff an easy access to medical analysis results. Such an analysis can be seen as a task, they are created by a doctor who then has to wait for the result. Also the shopping list example of Section 3.1 is currently under discussion together with the WNEC Lab. at Taiwan Tech for pushing it further and providing an implementation together with a mashup application.

We believe in a future where things, synchronous services, data, but also decomposable delayed services, coordinating business process, and so on, are unified within a consistent and seamless structure. We believe that this seamless structure is RESTful for its simplicity and scalability. The low entry barrier for constructing quick (and dirty) mash-ups as well as the use of well-established standards are further arguments for a world of RESTful services. Some starting point for such a world is provided in (Haller, 2010) and (IOT-A, 2012) but much remains to be done, like giving precise definitions and larger classifications, but we think that the present paper contributes to a small part of solving this complex puzzle.

References

D. Briskorn. *Sports leagues scheduling: models, combinatorial properties, and optimization algorithms*. Lecture notes in economics and mathematical systems. Springer, 2008. ISBN 9783540755173.

W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin. prest: a rest-based protocol for pervasive systems. In *Mobile Ad-hoc and Sensor Systems, 2004 IEEE International Conference*, 2004.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2, May 2002. ISSN 1533-5399.

B.L. Golden, S. Raghavan, E.A. Wasil, and Inc NetLibrary. *The vehicle routing problem: latest advances and new challenges*. Springer, 2008. ISBN 0387777784.

J. Gregorio and B. de Hóra. The atom publishing protocol. RFC 5023 (Proposed Standard), October 2007.

Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. Towards physical mashups in the web of things. In *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, June 2009.

Dominique Guinard, Mathias Mueller, and J Pasquier. Giving rfid a rest: Building a web-enabled epcis. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, 2010a.

Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the web of things. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, 2010b.

Stephan Haller. The things in the internet of things. Technical report, SAP, 2010.

Yinghui Huang and Guanyu Li. A semantic analysis for internet of things. In *Proceedings of the 2010 International Conference on Intelligent Computation Technology and Automation - Volume 01*, ICICTA '10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4077-1.

Tony Hürlimann. Reference manual of lpl. URL `http://www.virtual-optima.com/download/docs/manual.pdf`.

Thomas Luckenbach, Peter Gober, Stefan Arbanowski, Fraunhofer Fokus, Andreas Kotsopoulos, Kyle Kim, Samsung Advanced, Technology Sait, and P Box. Tinyrest - a protocol for integrating sensor networks into the internet. In *in Proc. of REALWSN*, 2005.

Carsten Magerkurth, Klaus Sperner, Sonja Meyer, and Martin Strohbach. Towards context-aware retail environments: An infrastructure perspective. In *Proceedings of Mobile Interaction in Retail Environments*, MIRE '11. DFKI, 2011.

Sonja Meyer, Klaus Sperner, Carsten Magerkurth, and Jacques Pasquier. Towards modeling real-world aware business processes. In *Proceedings of the Second International Workshop on Web of Things*, WoT '11, pages 8:1–8:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0624-9.

Toth Paolo and Daniele Vigo. *The Vehicle Routing Problem, SIAM Monographs On Discrete Mathematics and Applications.* Society for Industrial & Applied Mathematics, 2002.

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2.

L. Richardson and S. Ruby. *RESTful web services.* O'Reilly Media, Inc., 2007. ISBN 0596529260.

Klaus Sperner, Sonja Meyer, and Carsten Magerkurth. Introducing entity-based concepts to business process modeling. In Remco Dijkman, Joerg Hofstetter, Jana Koehler, Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, and Clemens Szyperski, editors, *Business Process Model and Notation*, volume 95 of *Lecture Notes in Business Information Processing*, pages 166–171. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-25160-3.

Erik Wilde and Alexandros Marinos. Feed querying as a proxy for querying the web. In *Proceedings of the 8th International Conference on Flexible Query Answering Systems*, FQAS '09, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04956-9.

'LPL Web Solver, [Online] Available: `http://diuflx71.unifr.ch/lpl/mainmodel.html` *(last visited on August 4, 2011)*

Internet of Things - Architecture, [Online] Available: `http://iot-a.eu` *(last visited on August 4, 2011)*

The WebSocket API, [Online] Available: `http://dev.w3.org/html5/websockets/` *(last visited on August 4, 2011)*

Atmosphere Comet Framework, [Online] Available: `http://atmosphere.java.net/` *(last visited on August 4, 2011)*

XMPP Standards Foundation, [Online] Availabe: `http://xmpp.org/` *last visited on January 10, 2012*

RFC 6120: Extensible Messaging and Presence Protocol (XMPP): Core, [Online] Availabe: `http://xmpp.org/rfcs/rfc6120.html` *last visited on January 10, 2012*

RFC 6121: Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence, [Online] Availabe: `http://xmpp.org/rfcs/rfc6121.html` *last visited on January 10, 2012*

RFC 6122: Extensible Messaging and Presence Protocol (XMPP): Address Format, [Online] Availabe: `http://xmpp.org/rfcs/rfc6122.html` *last visited on January 10, 2012*

Firefighter, Wikipedia, [Online] Availabe: `http://en.wikipedia.org/wiki/Firefighter` *last visited on January 10, 2012*

Firefighter Roles, [Online] Availabe: `http://www.esfrs.org/stations/roles.shtml` *last visited on January 10, 2012*

Role Structure in the British Fire Service, [Online] Availabe: `http://www.firesafe.org.uk/role-structure-in-the-british-fire-service/` *last visited on January 10, 2012*

Note

[1]http://www.flickr.com
[2]http://www.twitter.com
[3]http://maps.google.com
[4]http://www.datamed.ch
[5]Commission for Technology and Innovation