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

Andreas Ruppen, Prof. Dr. Jacques Pasquier
Software Engineering Group
University of Fribourg
1700 Fribourg, Switzerland
Email: {firstname.lastname}@unifr.ch

PD Dr. Tony Hürlimann
Decision Support Group
University of Fribourg
1700 Fribourg, Switzerland
Email: 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 tackles the problematic of RESTifying decomposable delayed services in a generic way. Secondly, it proposes a general architecture and validates it with a case study including a prototypical implementation.

***Keywords*-Web of Things, Internet of Things, RESTful architecture, Decomposable delayed Web-Services, WebSocket**

I. 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 production, shipment 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. Another definition can be found in [1] and [2]. By looking closer at the classification in [1], 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 [3] and RESTful services [4]. Indeed, RESTful interactions have several advantages over fully fledged WS-* web services [5]–[8]. Indeed, 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 photo service or Twitter, but what about cleanly integrating *decomposable delayed services*? In opposition to services like Google Maps, 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. To achieve this goal, we first present a motivating example where delayed interactions and service decomposition happens and we show what are the problems and challenges of integrating such services within a RESTful perspective. We will then present a general architecture which address these problems. In the last part we illustrate our architecture applied to a case study, including a prototypical implementation.

II. 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.

A. *Problematic*

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 parcel delivery example from Subsection II-B shows 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 four categories, each of them having its properties and special requirements regarding the architecture to choose in order to integrate it.

The first category includes the *short living* services. Even though some computation is 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. Such services integrate seamlessly into the Web of Things. Consuming them is just a matter of a *GET* request. In fact, a user cannot distinguish if the answer from the server is actually computed or if he has accessed a static resource.

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* [9]. 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* [10] standard being one of the most promising. A WebSocket opens a new TCP/IP connection in the browser. This 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 the first 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.

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 [11]. 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. [4]. Instead of requesting the resource with a `GET` statement, a *new task* is created. 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 Subsection II-C. 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.

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 example of Subsection II-B, 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 [4] must be further refined.

B. Motivating example — Parcel delivery center

We now come to an example application motivating the value of decomposable delayed services. We first present a concrete example and move then on to the abstract case. The presented example will be used as a use-case throughout the remaining of the paper. Consider a parcel delivery center which has to deliver parcels to its customers. Each parcel has an assigned priority adding a constraint on the delivery time. All parcels received during the day, will be distributed the next day respecting the given priorities. Upon arriving at the central, each parcel is scanned (for example by reading its RFID tag or its bar-code). Through this scanning, the systems knows the priority and the delivery address of each parcel. It is then added to the current distribution list. In the evening all parcels have to be assigned different tours.

This is an instance of the well known problem in operations research called *Vehicle Routing Problem with Time-Windows (VRP-TW)* [12]. 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.

C. Generic Architecture

Section II-B introduced a motivating example of decomposable 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 facade to an arbitrary delayed service.

Basic RESTifycation of the Resources: 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. [4]. To our point of view,

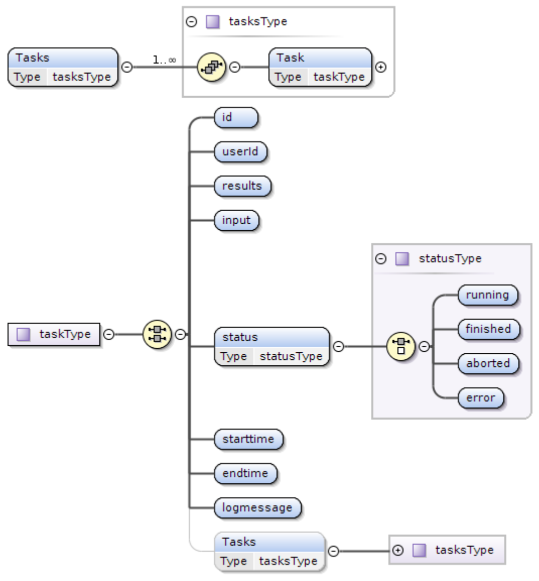

Figure 1. Conceptual overview of the tasks hierarchy

it is a good reasoning, but not enough. What happens for example, if the created task decomposes itself into several sub-tasks (see the motivating example in Section II-B)? 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 [4], there are not that many possibilities to bring asynchronous services into the world of RESTful services. Each request to a service 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. As Richardson et. al state 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 that service and *http://.../servicex/tasks/id/* points to a given task. This definition can be extended recursively; *http://.../servicex/tasks/id/tasks/* is a list of sub-tasks whereas *http://.../servicex/tasks/id/tasks/id/* points to one sub-task. This recursive definition can be drilled down to as many layers as needed. Figure 1 gives a conceptual overview of this hierarchy of tasks.

A deeper analysis of the example of Section II-B shows that the 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 a 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 log message are not directly part of the task. However, it is interesting to have access to this information.

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 1, tasks are defined recursively. A task can have several sub-tasks. This relation has to be translated into the system.

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 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: XML/JSON for simple tasks and multi-part form data for file upload. The multi-part creation comes handy when the description of the computation is complicated or when the service interaction with the service is file based (see Section III 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 task or a list of tasks. In the first case, it returns both the attributes of the task and a list of links to its sub-tasks. In the second case, it returns just a list of tasks. 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 [13] feeds to which users can subscribe to get, for example, all unfinished tasks or all tasks started before a given time [14].

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. In the case of an error, the log 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. 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.

Software Design Considerations: When building a RESTful delayed service, the decision of RESTifying the current service or building a new service from the 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 facade is created. Figure 2 shows a service with a WS- (or RPC) interface. The facade 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 facades. All the

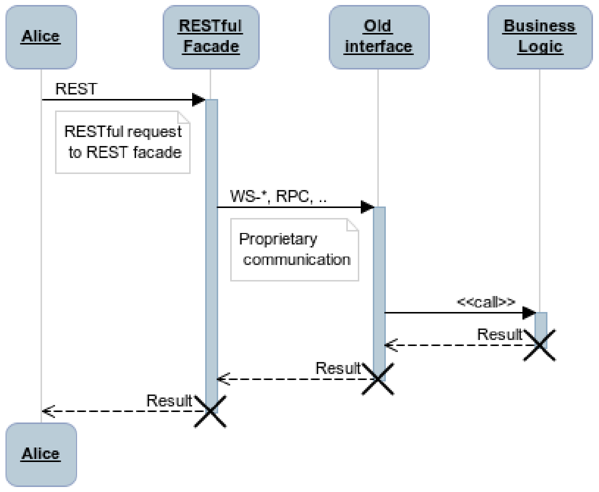

Figure 2. RESTifying a service

business logic remains in the original service. Therefore, the facade 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.

III. Prototypal Implementation

The parcel delivery central of section II-B 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 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 first computes the partitioning of the customer on the k vehicles and then define the best routing for each of them.

Suppose our central has to deliver 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” means that the customer must be visited not later than 9h25, “normal” means not lather than 11h05 and “slow” can be delivered the whole day till 16h00.

Figure 3 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 route 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.

To validate our findings of Section II-B, we have implemented the above scenario in a RESTful *decomposable delayed* service. Currently there is a program called LPL [15], [16] for solving the VRP-TW. Yet, any other solver capable of solving this problem could be used. LPL is only available as a Windows DLL; thus, it cannot be used directly from a mashup. We applied to this DLL our generic solution to RESTify it. Optimization problems, as in the case study, are described by models and their associated data on which the computation will be done. 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 facade 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 facade for it.

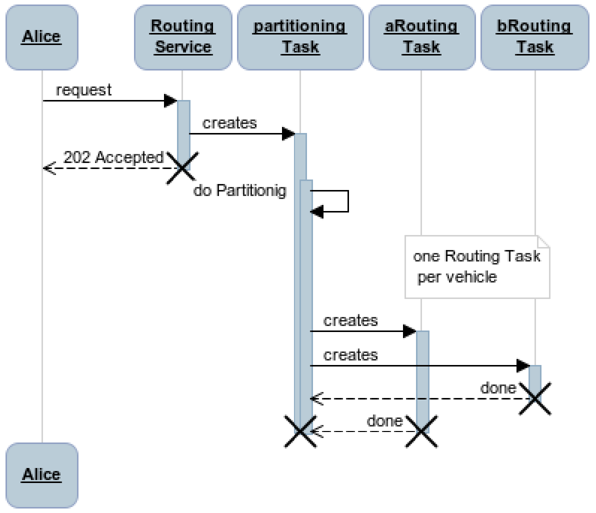

Figure 3. Requesting the “routing service”

All the methods mentioned in Section II-C were implemented. 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 against this data set. This is the first part of the computation where the partitioning of the parcels is computed. As soon as this first step - partitioning into four tours - has finished, the task attributes are updated, reflecting the new state as well as the result. The partitioning computation took 10 minutes. 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.

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. As stated before, the result is a list of links pointing to the individual parts of the solution. This request, as RESTful principles propose, delivers either a fully fledge HTML page or XML/JSON data which can then be used to be integrated into a mashup on the postmen smart-phones.

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 implemented facade contains only code interacting with the actual LPL solver and generating the recursive tasks. No business logic is implemented in the facade. Furthermore, the attributes discussed in Section II-C are sufficient to describe the system. They add enough information to ensure that the service is working, but not more than needed. 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 elsewhere. Furthermore, nothing stops us to integrate technologies like WebSockets, a Comet framework or simple polling to embellish the final application and provide more usability.

IV. 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 facade approach that such a service can be RESTified even if you are not its owner.

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) mashups 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 [1] and [2] 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

[1] S. Haller, "The things in the internet of things," SAP, Tech. Rep., 2010.

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

[3] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Technol.*, vol. 2, May 2002.

[4] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, Inc., 2007.

[5] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big"' web services: making the right architectural decision," in *Proceeding of the 17th international conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008.

[6] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards physical mashups in the web of things," in *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, Jun. 2009.

[7] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, 2010.

[8] D. Guinard, M. 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, 2010.

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

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

[11] D. Briskorn, *Sports leagues scheduling: models, combinatorial properties, and optimization algorithms*, ser. Lecture notes in economics and mathematical systems. Springer, 2008.

[12] B. Golden, S. Raghavan, E. Wasil, and I. NetLibrary, *The vehicle routing problem: latest advances and new challenges*. Springer, 2008.

[13] J. Gregorio and B. de Hóra, "The atom publishing protocol," RFC 5023 (Proposed Standard), Internet Engineering Task Force, Oct. 2007.

[14] E. Wilde and A. Marinos, "Feed querying as a proxy for querying the web," in *Proceedings of the 8th International Conference on Flexible Query Answering Systems*, ser. FQAS '09. Berlin, Heidelberg: Springer-Verlag, 2009.

[15] T. Hürlimann, *Mathematical Modeling and Optimization*. Kluwer Academic Publishers, 1999.

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