# A Computational Space for the Web of Things

Simon Mayer
Inst. for Pervasive Computing
ETH Zurich
Zurich, Switzerland
mayersi@inf.ethz.ch

David S. Karam
Chair of Scientific Computing
TU Munich
Munich, Germany
karam@in.tum.de

## ABSTRACT

The expansion of the World Wide Web to include information that is generated by physical devices with embedded sensing and actuation capabilities entails a surge of high-frequency real-time data that is mostly published without further processing in its raw form. To derive "smart" decisions from this data and thus use it to enable a "smart world" requires the distilling of more abstract, higher-level knowledge from it. In this paper, we propose the concept of a *computational marketplace* as a framework to enable the analysis and aggregation of real-time data. Here, multiple tiers of hyperlinked algorithms from different providers interact to refine data within computational graphs, which are linked structures of cascaded processing steps. We present an analysis of the key constraints on such a framework and provide a corresponding implementation as well as results from evaluations in an experimental use case scenario.

## 1. INTRODUCTION

A central concern in the field of Pervasive Computing and specifically in the Web of Things (WoT) domain is to expand the World Wide Web to also include content that is generated by real-world things such as physical devices and their sensors and actuators. We expect that this data will act as an enabler of a "smart world" by helping humans and machines make smarter, faster, and more well-informed decisions. However, raw data produced by machines does not allow us to directly and consistently make the smart decisions we hope for. First, devices produce real-time data at much higher rates than those of humans. Second, the data is usually published in its unprocessed form (e.g., temperature values) rather than as more abstract information (e.g., weather conditions). So, while real-time data from physical devices contains valuable information, it has to be further processed, aggregated, and analyzed to be used within decision-making processes. Research in Pervasive Computing and in the WoT domain has thus given us the ability to access an overwhelming flow of raw, real-time data but with no clear method of consistently deriving higher-level knowledge from it.

In this paper, we propose an open computational marketplace that is designed to enable the distilling of information from raw data via the interaction of linked algorithms. The marketplace shall allow clients to link their data feeds to algorithms offered by competing providers. The algorithms themselves are hyperlinked to enable a multi-tier model of computations, where data is refined by computational agents in cascaded processing steps.

## 2. MOTIVATION

The real-time Web has so far been driven by three basic trends (cf. Table 1). On the first level, a *proliferation of hardware devices* has enabled data to be produced in real time, either user-driven via mobile applications on smartphones or automatically by the use of physical devices that embed sensing and actuation capabilities. The resulting availability of data has in turn led to the development of *information markets*, platforms which aggregate that information and offer it to others. On the third level, the abundance of such data has enabled the development of *computational markets*, where different services distill a refined interpretation of that raw information. This trend can be seen in the human Web, where global *information markets* such as Twitter and Google increasingly influence our personal and business lives. Tools for building a computational market on top of the human Web are also emerging: Lithium[1], for instance, helps companies to identify their most enthusiastic customers and thereby to better define, market, and sell their brands by mining social data feeds.

The same trend is visible in the world of machines: Starting with the widespread use of smartphones, we have seen a

[1] www.lithium.com

| | **Human Web** | **Web of Things** |
|---|---|---|
| **Commodity Hardware** | Smartphones | Smart consumer products, Smart Meters, Arduinos |
| **Information Markets** | Twitter, Facebook, Google | Pachube, Sen.se |
| **Computational Markets** | Social data mining | *Computational Marketplace* (?) |

**Table 1: Major representatives of *commodity hardware*, *information markets*, and *computational markets* in the *Human Web* and the *WoT*.**


*WoT 2012*, June 2012; Newcastle, UK

proliferation of *commodity hardware* equipped with sensing and communication capabilities. Tools like Arduinos and Sun SPOTs allow researchers and hobbyists to build and deploy Web-enabled devices that bring with them sensing and actuation capabilities for the real world. "Smart" products like Ploggs[2] or the Withings Scale[3] give end users the ability to monitor and analyze their devices' electricity consumption or even their own weight and share and compare the acquired data with their peers. This trend gives rise to open real-time *information markets* like Pachube[4] or Sen.se[5] that consume data acquired by sensing hardware and allowing users to retrieve, visualize, and filter it in real time. Data that is produced using commodity hardware and published via information markets is increasingly getting processed and used in a crowd-sourced fashion. As an example, consider the "Wind from Fukushima" application that was created after a tsunami caused a number of nuclear accidents at three reactors in the Fukushima nuclear power plant complex in Japan. This application mashed the user's location with the latest radiation readings and wind speed and direction to help users determine whether they were in danger. This application is an inspiring example for an emerging computational market on top of real-time, real-world data. The (Android) market through which it was published however is a closed system where applications cannot make use of computations done by other applications.

Our goal is to retain the ability to arbitrarily mash raw data along with refined information, allowing us to incrementally increase the capabilities of Web-based systems. In our perspective, this requires multiple open tiers of computations, where successive refinements are produced as output and fed as input to other algorithms: Algorithms should be linked together in an open and extensible fashion in a construct that we call a *computational marketplace*.

## 3. RELATED WORK

The described computational marketplace should offer its services to clients while being able to scale to the size of the Web. The two main technologies for creating such an architecture in the context of the Web are WS-* Web Services and Representational State Transfer (REST) [5]. Work done to compare these two approaches especially with respect to Internet of Things applications regarding their performance, features, and usability aspects [6, 9] concludes that RESTful Web services are very attractive for scalable and open applications. For this reason, REST is the architecture of choice for the design of our computational marketplace.

Our goal is not to create service offerings in isolation, but to aggregate computations in a distributed fashion and reach a form of collective intelligence. A very early effort to create such an open platform was *Yahoo! Pipes*[6], which provided an engine for aggregating and filtering *Really Simple Syndication* (RSS) feeds. The focus of our provisioned computational marketplace is similar, however, our target group are providers of algorithms rather than RSS feed providers. Another service offering platform with a different architectural style can be seen in the *Google Apps Marketplace*[7], where developers can write software and offer and run it via the Google infrastructure. The various marketplaces for mobile phone apps also offer a way where developers can sell the results of their computations to others. These can, however, not be seen as enablers of an open computational framework as they do not offer support for mashing up computations.

The problem of providing such mashable computations has also been approached in the context of the Web: JOpera [8], a composition tool for RESTful Web Services, provides a visual language for defining both control and data flow graphs, and an execution engine for the designated workflow. Bite [10] proposes a similar BPEL-inspired services composition language describing both control and data flow and partially supports an HTTP-based uniform interface, dynamic typing, and state inspection. In both JOpera and Bite, the composition workflow is exposed as a unique resource in itself. These engines however do not only harbor the links to the computations but also invoke the computations themselves which inhibits the scalability of the system. Furthermore, these composition engines do not completely adhere to the "hypermedia as the engine of application state" (HATEOAS) constraint that is central to the REST architectural style. Finding that not enough emphasis is placed on the hypermedia characteristics of RESTful models, the authors of [3] use a Petri Nets model with a layer defined by the *Resource Linking Language* (ReLL) [1], extending the execution engine with meta-information on how the resources are linked together which is reusable in scenarios such as crawling [2] and semantic integration. The authors of [4] put more emphasis on linking as an integral part of composing resources on the Web and guiding RESTful machine-to-machine interaction. By providing links to further resources in the composition, the resources themselves act as enablers of the HATEOAS constraint without the need of a central choreographer. The concept of placing importance on resource linkage is also gaining traction within industry, with new emerging REST frameworks (e.g., Restfulie [7]) that are built with the requirement of satisfying the HATEOAS constraint as their primary design goal. Hyperlinked computations centered around the HATEOAS constraint will also be central to the following derivation of constraints for a Web-based computational marketplace.

Finally, we mention ProgrammableWeb[8], a directory of Web APIs and mashups and one of the most prominent Web-based mashup platforms. ProgrammableWeb currently hosts information about more than 5000 APIs (e.g., text analysis services, restaurant guides) and more than 6000 mashups that provide aggregation, analysis, and visualization services on top of these APIs. Additionally, ProgrammableWeb offers capabilities for developers to comment and rate the APIs and mashups in an effort to foster a community around the creation, provisioning, and usage of Web-driven data processing tools. This outlines a clear trend: Developers are not only willing to perform remote aggregate computations, they also wish to expose the result of those computations for use by third parties.

## 4. CONSTRAINTS

The computational marketplace as we defined it needs to be able to openly and arbitrarily link algorithms, retaining scalability, fault tolerance, and change tolerance. In this

[2] `www.plogginternational.com`
[3] `www.withings.com`
[4] `pachube.com`
[5] `open.sen.se`
[6] `pipes.yahoo.com/pipes/`
[7] `www.google.com/enterprise/marketplace/`
[8] `www.programmableweb.com`

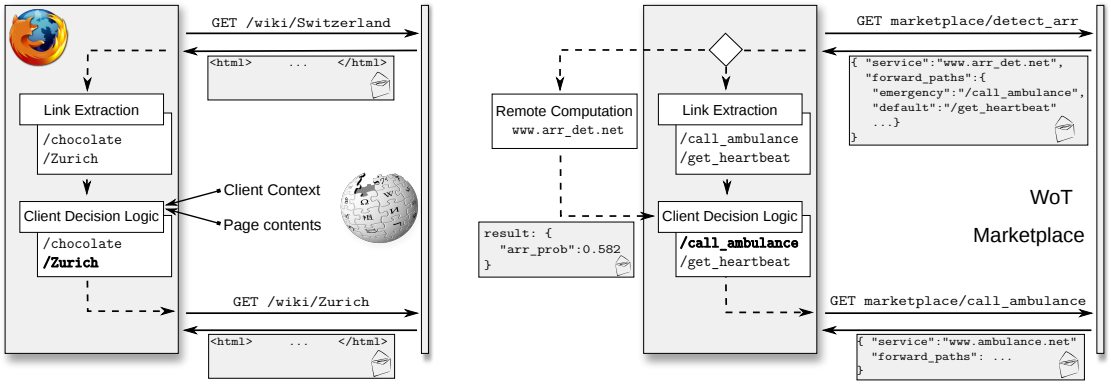


**Figure 1: a) A client browsing Wikipedia. b) A client using the computational marketplace.**

context, fault tolerance would allow clients to keep executing computations even when certain computational providers fail to deliver for their application needs. Change tolerance would allow clients to retain their ability to perform computations even when the servers on which the computations are hosted change the way they do theirs. ProgrammableWeb acts as a directory, allowing us to openly find algorithms, but not link them. Existing mashup engines like Yahoo! Pipes and JOpera allow us to arbitrarily link algorithms, but due to their centralized nature, not in an open, scalable, and fault tolerant fashion. It thus becomes evident that a different architecture has to be designed to achieve these desirable properties within a single system. In an attempt to leverage the Web's decentralized model for helping to guarantee these properties and to apply it to a computational marketplace, we now derive a set of architectural constraints on an open N-tiered computational marketplace on top of the Web.

## 4.1 Path Traversal Guidance

A central constraint in the design of the REST architectural style is HATEOAS, which requires that servers, while providing representations of their state to clients, also provide links that clients can follow to invoke state transitions. In that sense, the server can be considered to *guide* the client through its traversal of, for instance, a website: As an example, consider a human user browsing Wikipedia and requesting the wiki page about the item *Switzerland* (cf. Figure 1). The server will return an HTML page to be rendered on the client machine and displayed to the user. Contained in the page are multiple links to further websites (e.g., *chocolate*, *Zurich*) which again are displayed appropriately (e.g., as being "clickable"). Now, based on factors like the page content and the human user's context (e.g., goals or prior knowledge), the user will select one of these links to follow and thereby initiate a state transition. An interface change is thus transparent to the client, who is only waiting for controls and links that guide it through possible state transitions and is thus tolerant to changes in the hypermedia structure.

The computational marketplace should thus be organizing the composite computation by harboring the possible computational paths while leaving the decision of which concrete path is taken to the traversing entity. For instance, consider a client that uses an arrhythmia detection algorithm via the computational marketplace (cf. Figure 1). The client first requests information on the *detect_arrhythmia*-node by querying the marketplace and gets, as response, both the URI of the arrhthmia detection algorithm and the *forward paths* (i.e., links to information about further computations) that it may follow after getting the result from the arrhythmia algorithm. The client then invokes the remote computation and – just like the human user before – uses its result as well as internal parameters (i.e., its *context*), to select the path to follow. In this manner, modifications of the paths in the computational marketplace (e.g., adding or removal of intermediate computations) remain transparent to the traversing entity.

## 4.2 Interface Discovery

Computational providers require a way of advertising their composite services to make them discoverable by clients. Centralized models to describe and discover services like, for instance, the Universal Description Discovery and Integration registry (UDDI), rely on service registration for discovery which works well in closed scenarios. For a decentralized, hyperlinked system like the Web, however, having discovery performed by crawlers which traverse resources by following links is more applicable. The topology of a computational marketplace should thus be inferred from references rather than being constructed by registration. Discovery in a computational marketplace should be enabled by the fact that the algorithms are hyperlinked as soon as they are involved in a composite computation.

## 4.3 Self-Similarity & Statelessness

We claim that a marketplace must enable the combination of computational agents that perform different types of computations as well as dynamic expansion and contraction with respect to computational agents that perform the same type of computation. The marketplace should thus offer *horizontal flexibility*, which refers to the ability of mashing up modules into combined modules that exhibit the same structural properties, allowing them to be further combined with more modules. Constructs at the heart of a computational marketplace must therefore exhibit *self-similarity* to ensure a scale-invariant structure. Second, it should offer *vertical flexibility* to allow the adding and shedding of computationally equal agents. Computational agents in a computational marketplace must therefore execute *stateless* computations, meaning that all information relevant for a certain computation must be passed along with the computation request. If a computation requires state, then it must be adressable by a URI which is passed along with the computation request such that issues like load shedding and state migration can-

| Constraint | Scalability | Fault Tolerance | Change Tolerance |
|---|---|---|---|
| Self-Similarity & Statelessness | +++ | + | ... |
| Traversal Guidance | + | + | +++ |
| Path Optimization | ++ | + | ... |
| Interface Discovery | ++ | ... | + |
| Security & Billing | ++ | ... | ... |

**Table 2: Overview of the derived constraints (+++: Key enabler, ++: Pronounced effect, +: Weak effect).**

not limit the marketplace's scalability. This property is also key to the critical aspects of load balancing, recovery from failure, and dynamic resource allocation. It is also key to the concept of the computational marketplace as a *market* for computations, as clients should be able to switch freely between providers of the same kind of algorithm.

## 4.4 Computational Paths Optimization

If the latency of an algorithm used by a client increases, the client could wish to switch to another provider offering the same computation (but probably of lower quality). We argue that a computational marketplace should offer a mechanism that allows clients to optimize computational paths according to their application needs, for instance with respect to the speed, money cost, or accuracy of the required computations. The marketplace should therefore provide information related to these qualities of computational paths. As the clients need to retain control over which paths to traverse, it is however not required of the marketplace to *decide* which paths are optimal for which clients.

## 4.5 Security and Billing

Computational resources often represent an expensive utility that is secured by authentication and authorization schemes to restrict its use to an exclusive set of customers, who may also be billed based on their usage of the system. Especially for Web APIs, tasks related to security and billing are often handled within a centralized model where clients register in exchange for credentials (e.g., an API key) required to use a service. We argue that the tasks of security and billing should not be handled by the marketplace itself, but rather propose that algorithm providers incorporate a third-party billing and security scheme (e.g., OAuth 2.0[9]) to allow the delegation of resource invokations on behalf of the client.

# 5. A COMPUTATIONAL MARKETPLACE IMPLEMENTATION

In this section, we present our implementation of a computational marketplace that has been developed to satisfy the derived constraints. We describe its components and their correspondence to the constraints defined in Section 4. For reference, Table 2 summarizes the constraints together with our expectations regarding their influence on the properties of scalability, fault tolerance, and change tolerance. Figure 2 gives an overview of the marketplace ecosystem, which itself is a RESTful service designed to conform to the requirements for RESTful services composition (cf. [8]). A computational marketplace is responsible for harboring the computational graph whose nodes represent computations and whose edges represent paths between computations and for handling issues such as load balancing, path optimization, space expansion and contraction, and guidance of the traversing entity. For *providers* of algorithms and data streams, the system offers the functionality to publish their algorithm APIs along with API descriptions to support clients in using their services. *Mashup developers* may use the marketplace to define computational graphs on top of the provided algorithms and thereby create $N$-tiered computations on data supplied by themselves or others. To guide clients along the paths of a newly created mashup, developers define subgraphs (represented by the marked nodes in Figure 2) that clients can request to execute the computations associated with a given mashup. Finally, *clients* execute composite computations by having the marketplace guide them through the graph structure using the provided linkage information. They themselves invoke computations on separate servers in a distributed manner.

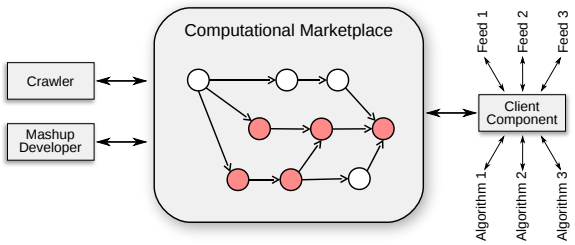


**Figure 2: The marketplace ecosystem.**

## 5.1 Coordination Component

The Coordination Component is responsible for the management of all computational mashups and provides APIs for algorithm providers, application developers, and clients who wish to use the composite mashups.

First, it provides the possibility for algorithm providers to register the computations they offer. Each computation entry point is modeled as a *vertex resource* and can be established by developers creating applications on the marketplace. These developers can also establish routes for computations by creating *link resources* on the marketplace which imply that the involved computations are related and that the marketplace should signal a client that performed the first that a related computation exists on its path.

Second, the Coordination Component allows adding transformation maps to the link resources, specifying how one algorithm's output should be mapped to another algorithm's input. We have introduced transformation maps to provide a certain degree of automation, and thus convenience, for mashup developers for simple and recurring mapping tasks (e.g., downsampling of data, renaming of data items, simple type conversions). However, for more complicated transformations, transformation algorithms should be exposed and linked to the "worker" algorithms, which in turn can lead to the creation of reusable transformation components.

Finally, clients use the Coordination Component to find a path from their current computation to the next within the context of a selected mashup. The Coordination Component also provides information to clients regarding various cost metrics (i.e., latency, accuracy, or money cost) within their computations. For instance, clients may ask for the fastest or the cheapest computational paths that will allow them to execute a specific composite computation. This in

[9] `tools.ietf.org/html/draft-ietf-oauth-v2-22`

turn may cause these paths to slow down or become more expensive which will give clients the opportunity to adapt their traversal to account for these changes (cf. Section 6.2).

## 5.2 Graph Crawler

From the view of the marketplace, the crawler is an external component. It is, however, the responsibility of the marketplace to offer an interface that allows crawlers to enter the computational graph and discover new vertices (cf. Section 4.2). Our implementation enables this via the Coordination Component, which, for each *mashup graph* resource, exposes the graph's entry vertex.

## 5.3 Security and Billing

Since computational providers are themselves responsible for authorizing and billing their users, there is no component in the marketplace to take care of the issue of security and billing. However, the provisioning of this functionality can be inlined as a computational vertex in a graph just like other computational agents. As a demonstration of this, the prototype mashup presented in Section 6.1 includes a vertex which directs the client such that an OAuth-based exchange with the marketplace is initiated.

## 5.4 Client Component

In analogy to the browser, which is the "client component" of the human Web that communicates with the server on behalf of the human user to parse and render Web content graphically, the Client Component is responsible for handling responses from the Coordination Component that indicate possible vertices to follow along the computational path. In particular, its task is to select one of the links on behalf of the client and thereby direct the computation towards the corresponding vertex and its associated algorithm. Because the process of selecting an appropriate state transition needs to be automated for machines, our implementation contains *Traversal Managers* that the user can program to traverse the graph in a specified manner. To give an example, in our prototypes we make use of simple traversal managers that, for instance, always select the first link in the list of possible forward paths or always select the path that carries the lowest latency as announced by the Coordination Component. The concrete implementations of the traversal managers, however, are fully decoupled from the marketplace and have to be written by the developer wishing to use the marketplace, where usually also the results from preceding computations are taken into account. The marketplace itself cannot know how the decision making process should be designed.

# 6. EVALUATION

In this section, we show a concrete example of a computational mashup that is based on the marketplace implementation. We also provide an analysis of the dynamic optimization capabilities of the marketplace and show in an experimental scenario how it is able to balance client requirements and offerings from computational providers.

## 6.1 Arrhythmia Patient Scenario

As a proof of concept of the presented computational marketplace implementation, we have implemented a real-world scenario where raw and processed data are mashed up to reach the final result. The presented scenario represents an attempt to incorporate multiple real-world services into a single mashup, in an effort to come as close as possible to guaranteeing real-life applicability of our developed concepts. The only two services that were developed internally were the arrhythmia detector and the ambulance path optimization service. All other functionality, persistence, and visualization was performed using external tools. In the following description, we leave out the visualization and persistence algorithms for conciseness.

Four computational graphs form the major subcomponents of the mashup: The *Arrhythmia Graph* retrieves heartbeat data feeds and decides whether the patient is experiencing an arrhythmia. Its main components are the heartbeat feed[10] to retrieve a patient's heartbeat data and two arrhythmia detection algorithms. The *Weather Graph* retrieves raw weather data (temperature, humidity, etc.) from the Yahoo! Weather API and outputs one of 47 high-level descriptions of the weather conditions (e.g., "partly cloudy") using the Google Predict API. The *Traffic Graph* retrieves (simulated) geolocation information of cars and pedestrians and mashes it with the weather prediction to create traffic density estimates. Finally, the *Dispatch Graph* uses location information from simulated patients, ambulances, and hospitals and mashes it with traffic density estimates to dispatch ambulances to patients using the Bing and Google APIs to retrieve driving information. In the composite scenario, all these graphs are mashed up to incorporate refined weather and traffic information with the arrhythmia detection services to optimize the ambulance dispatch process.

To create the described computation in a formerly empty computational marketplace, a mashup developer first creates computational vertices via the Coordination Component by specifying, for each, the URI of the corresponding computational resource, the corresponding REST methods, and the output data type. Additionally, the request header, path, query, and form parameters may be specified if required. Next, the vertices have to be linked together while specifying the transformation mapping between the algorithms. The implemented transformation maps do a simple object-field mapping which, apart from piping outputs from one algorithm into another, also allows to aggregate the outputs of multiple algorithms as inputs for a single computation. If a client arrives at the vertex corresponding to the Google Predict API for classification of the weather conditions, the marketplace redirects the client for authorizing the use of his/her Google Predict account via OAuth.

## 6.2 Marketplace Exchange

We next present the results from our evaluation of the dynamic properties of the marketplace where we specifically highlight the optimization component which allows clients to dynamically switch between providers according to their computational needs with respect to a computation's money cost, its duration, and/or its accuracy. We used a simple computational graph that consists of three dummy arrhythmia detection algorithms which all exhibit time and money costs that grow linearly with the number of clients each is servicing at a certain point in time. The Coordination Component delivers the URIs and cost functions of these algorithms on every client request and the clients decide to route their computations over the provider that best

[10]Samples from the database on malignant heart rhythms from the MIT-Beth Israel Hospital

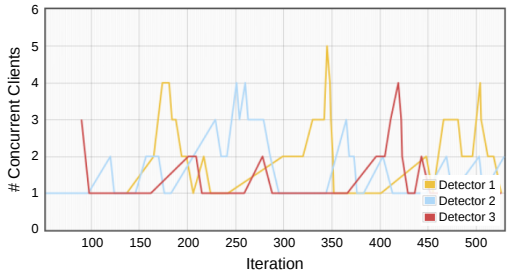


**Figure 3: Number of concurrent clients over time with fifteen clients on multiple optimization metrics.**

suits their needs, where different clients have differing preferences regarding the three metrics. So, the marketplace tries to achieve a dynamic equilibrium by decentralized decision making, where clients' needs are satisfied by the offerings of various providers: While each client acts in a personal manner, trying to optimize its own needs, it affects the provider offerings, hence affecting other clients which are also attempting to optimize their computations.

We present a test case where fifteen different clients attempt to use the mentioned arrhythmia detection algorithms. Three of the clients always choose the same detector (each chooses one of the available three) while seven clients always request the *cheapest* computational path and five clients request the *fastest* computational path. Figure 3 shows the number of clients that are being serviced by each detector in this scenario over time. We can observe the described coordination of demands by the marketplace, although the situation is a bit complicated as multiple clients and multi-dimensional metrics are involved. Whenever a peak load is reached (e.g., *Detector 1* at iteration 350), the marketplace recovers by routing the dynamic clients to other detectors, as indicated by the sharp drops in the number of concurrent clients shortly after the peaks occur.

We can see that the computational marketplace implementation is indeed able to balance client demands and provider offerings. However, an optimal solution – assigning all clients statically to reduce the overall waiting time – is not achieved as the design of the marketplace makes a concious trade-off on efficiency for a de-coupling of the clients and the marketplace component to ensure the scalability, fault tolerance, and change tolerance of the system

# 7. CONCLUSIONS AND FUTURE WORK

In this paper, we have described our implementation of a computational marketplace for the WoT, an open decentralized computational framework that supports the crowd-based distilling of information from raw data streams. We have defined constraints that allow to leverage the Web's proven architecture to guarantee properties like scalability, fault tolerance, and change tolerance in such a setting and have proposed a REST-based implementation of a computational marketplace where computational providers compete in their provisioning of algorithms for analyzing clients' data and where the results of computations may subsequently be fed into other computational nodes in an $N$-tiered computational model. Finally, we have provided a proof of concept of a possible real-world scenario that can be achieved within the implemented framework and have analyzed the computational marketplace with respect to its abilities of balancing providers' offerings and clients' computational requirements.

We plan to use the developed computational marketplace as a testbed for patterns in the domain of interlinking algorithms, specifically with respect to providing mashup creation support. On a syntactical level, the marketplace could for instance propose nodes that would allow automatic linking on the basis of their API descriptions and allow developers to not only search by keyword, but also by required output (or input) type, similar to the Hoog$\lambda$e[11] API search engine. This concept could then be extended to include semantic technologies like ReLL [1] or RESTdesc [11] and thereby even allow clients to automatically reason about computations with the help of annotated computational paths. Technologies like these could, in the long run, also prove useful to facilitate the client's path decision process described in Section 4.1.

## Acknowledgements

This work was supported by the Swiss National Science Foundation under grant number 134631.

# 8. REFERENCES

[1] R. Alarcón and E. Wilde. Linking Data from RESTful Services. In *Proc. LDOW*, 2010.

[2] R. Alarcón and E. Wilde. RESTler: Crawling RESTful Services. In *Proc. WWW*, 2010.

[3] R. Alarcon, E. Wilde, and J. Bellido. Hypermedia-Driven RESTful Service Composition. In E. Maximilien, G. Rossi, S.-T. Yuan, H. Ludwig, and M. Fantinato, editors, *Service-Oriented Computing*, volume 6568 of *Lecture Notes in Computer Science*, pages 111–120. Springer Berlin / Heidelberg, 2011.

[4] J. Bellido, R. Alarcón, and C. Sepulveda. Web Linking-based protocols for guiding RESTful M2M interaction. In *Proc. ComposableWeb*, 2011.

[5] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[6] D. Guinard, I. Ion, and S. Mayer. In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers' Perspective. In *Proc. MobiQuitous*, 2011.

[7] S. Parastatidis, J. Webber, G. Silveira, and I. S. Robinson. The Role of Hypermedia in Distributed System Development. *Proc. WS-REST*, 2010.

[8] C. Pautasso. Composing RESTful services with JOpera. In A. Bergel and J. Fabry, editors, *Proc. SC*, volume 5634 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2009.

[9] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *Proc. WWW*, 2008.

[10] F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf. Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. *IEEE Internet Comput.*, 12(5):24–31, 2008.

[11] R. Verborgh, T. Steiner, D. Van Deursen, R. Van de Walle, and J. Gabarró Vallés. Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In *Proc. NWeSP*, 2011.

[11] `www.haskell.org/hoogle`