An architecture for the Web of Things

Dave Raggett
W3C/ERCIM
2004, route des Lucioles
Sophia Antipolis - France
dsr@w3.org

ABSTRACT

The real World consists of physical things that computer software can interact with through sensors and actuators. This can be combined with rich descriptions, e.g. of devices, people, places and events. There is a huge potential for services, and most of the work to date has focused on communication technologies for the Internet of Things. It is now timely for an increased focus on how to create services, and the importance of standards for realizing the potential.

The *Web of Things* extends the Semantic Web into the real World for an open ecosystem of value added services based upon physical sensors, actuators and tags, together with rich descriptions of physical and abstract things. This paper presents an architecture for a scalable services platform for the Web of Things with APIs for services layered on top of HTTP and Representational State Transfer (REST).

Categories and Subject Descriptors

X.4 [**Insert**]: ACM Categories

Keywords

Internet of Things, Services, Cloud computing, REST, Semantic Web, Web Security, Privacy, Open Standards

1. WEB ARCHITECTURE

The Web is based upon open standards for addressing resources (URIs [1]), protocols for accessing resources (HTTP [2]), and formats such as HTML [3], CSS [4] and JavaScript [5], for more details, see W3C Web Architecture [6]. The Web has evolved from browsing static web pages to applications and services, with the browser hosting the user interface. The application processing is distributed: some occurs in the web page scripts executed in the browser, other processing occurs in the server.

The HTTP protocol is designed to support caching of resources to reduce the load on servers for static or infrequently changing content. Applications often involve dynamically generated pages tailored to a specific user's needs. This is typically data driven, involving database queries, and sourcing information from third party servers.

The distributed nature of clients and servers provides a degree of scalability compared to centralized approaches. Scaling to fulfil increases in demand for a given web site can be accomplished in a number of ways. The DNS servers can be configured to map a given domain name to multiple servers to spread the load. Front-end servers that handle the HTTP client requests can spread the computational load across multiple back-end servers.

Cloud computing refers to the means to fulfil compute or storage requirements by connecting a large number of computers. These computers are often hosted by third parties as a service at a lower cost that it would be to provision these yourself. Virtualization allows a single physical computer to be shared as a number of virtual computers. This gives users the ability to install and configure their own software as needed. Scalable solutions for data storage involve mechanisms for replication and synchronization. If most data queries are read-only, the mechanisms can be designed to speed up read-only access at the cost of slower update queries.

Graphics processing units (GPUs) are very efficient for matrix and vector operations compared to traditional computer processing units (CPUs). Cloud based GPUs [7] are expected to be increasingly important for speech and image processing, machine learning, translation, and emulation of neural networks [8].

For highly distributed solutions, some form of peer to peer computing is appropriate. This can be made more efficient if the number of peer connections per node follows a power law [9], so that any arbitrary pair of nodes can be connected to each other through a relatively small number of intermediaries [10]. Distributed hash tables (DHTs) allow you to distribute tasks across a large number of nodes through a mechanism that maps the task description to an index in an array. A related technique is to use Bloom filters which have the property of constant time to add items or to check whether an item is in the set, independent of the size of the set.

2. THE WEB OF THINGS

Bar codes have made it easy to scan identifiers for physical items. This has had a huge impact on tracking items through the supply chain or assembly line, or as they are transported across the World, as well as speeding checkout at a retail point of sales terminal. The year on year improvements in electronic circuits has made it practical to provide inexpensive radio frequency identification (RFID) tags. These are often more convenient than barcodes, e.g. not requiring direct line of sight to the scanner. RFID has evolved to support integrated storage, computation, and even simple sensors.

At the same time the communications technologies for sensors and actuators have evolved at a rapid pace. Low cost battery powered sensors are now available that operate for several years. Falling costs are making it practical to integrate sensors in all kinds of environments. Application areas include:

- Smart cities – transport, power, water, planning, ...
- Smart homes – home automation, extended warranties & proactive maintenance, security, entertainment, ...

WS-REST 2014, Seoul, Korea
ACM (to be determined).

- Home healthcare – improved outcomes and reduced costs
- Retail – stock taking, and richer shopping experiences
- Smart factories and distribution

Work on the Internet of Things [11] has mainly focused on communications technologies and much less so on how to create services. There are plenty of challenges: scalability, security, interoperability, coping with continuing change in communications technologies, a lack of clarity in respect to business models, privacy, personal control and threats to individual freedoms.

The Web of Things is a term coined in 2007 [12] for applying web technologies to blend the Internet of Things with the Semantic Web [13] to enable value added services.

- Interaction with the physical world through tags, sensors and actuators (the Internet of Things),
- Machine interpretable descriptions and context (i.e. the Semantic Web and linked data) and
- Open markets and ecosystems for services using scalable distributed cloud-based platforms.

The basic idea is to expose sensors and actuators via HTTP. The server involved hides the details of how the sensors and actuators are connected to it. This allows you to use whatever communication technologies make the most sense for the given environment. The sensors and actuators in many cases won't be using HTTP due to the need to conserve battery life. Some examples of communications technologies:

- WiFi, Weightless, ZigBee, CoAP & 6LoWPAN, NFC, BlueTooth & BLE, ANT, Infrared, USB, IEEE 1394, DASH7, KNX for buildings, EnOcean, GPRS/3G/4G, WiMAX, etc.

Web applications can directly access services exposed by HTTP, subject to the same origin security restriction. This limits web page scripts to accessing HTTP on the same server that provided the HTML web page. This is not a problem if the web applications are installed onto the same server as the server side scripts for accessing the sensors and actuators. Several companies have recently announced plans for open standards for services running on home gateways [14]

There are work arounds for the same origin restriction. One approach is to use the HTML script element to load data into the page's scripting execution environment. A related approach uses *document.write* to dynamically extend the web page. More recently, W3C has defined cross origin resource sharing (CORS [15]) where services use an HTTP header to indicate from which web page origins they are willing to accept requests.

3. OPEN MARKETS OF SERVICES FOR THE WEB OF THINGS

Small HTTP servers hosting services are at risk of being overloaded if a given service becomes very popular. A work around is to treat the server as a back-end server, and to publish the service using cloud-based proxy servers which are provisioned to match the demand. This is the approach being taken by the European project named Compose [16].

Value-Added Services

A cloud-based platform can be designed to support both scalability and easy registration of services. The starting point is a description of the service as a REST API. The platform also provides a basis for value added services that build upon other services:

- By combining multiple services in some useful way
- By combining services with rich descriptions and other sources of information
- By transforming service data into the same, higher or lower level of abstraction

Data fusion combines multiple sources of information, e.g. anonymised data on the locations of cars with map data to generate traffic density maps. An example of a transformation at the same level of abstraction is a conversion of data units such as temperatures from Celsius to Farenheight. Another example is a conversion of data formats, e.g. from XML to JSON. Biological metaphors for pipelined perception and actuation are applicable to tranformations at different levels of abstraction:

- Extracting meaning from progressive interpretation of ambiguous or noisy data
- Mapping high level intent to coordinated & synchronized control of actuators

Examples of perception include: speech recognition, face detection, tracking of pedestrians and cars in street scenes. Examples of actuation include: coordinating traffic lights based upon current traffic conditions, and signalling smart appliances to reduce the load on a electrical grid in response to peaks in demand.

Biological insights are also relevant to creating intelligent agents that act on their user's behalf, e.g. based upon a representation of goals, the ability to construct and execute plans, and to learn from experience. This is something where an interdisciplinary approach is likely to pay dividends, e.g. computational linguistics, cognitive science and artificial intelligence.

All this suggests the value of a service platform with support for a heterogenous mix of programming languages, e.g. simple expressions for data conversions, scripting languages for flexibility, rule-based languages for logical inferences, and specialized languages for exploiting GPUs for efficient operations on numeric data.

Markets of services involve people in a variety of different roles. Some people design and sell services. Others purchase services and apply them to their personal or business needs. The former is analogous to selling native apps through app stores for use on iOS and Android devices. As an example Peter sells a service for home security webcams. Mary purchases this as a packaged service, and applies it to her webcam in her garden. She needs to bind the service to her webcam via some kind of discovery/peering mechanism. She also needs an app to access the service to control the webcam and to view the images and video sequences it captures. She further has to decide whether to share the data with others, e.g. her friends, or even to make it publicly accessible.

Looking at this example in a little more detail, Mary's webcam supports taking video clips triggered by a motion detector. The video clips are uploaded to the cloud for storage. Mary can use the app to view a history of the video clips. She can register for notifications of activity, and request a direct live video feed together with the means to pan and zoom the webcam for a better view. What this shows, is that the webcam is a device with an API and is not just a simple data feed.

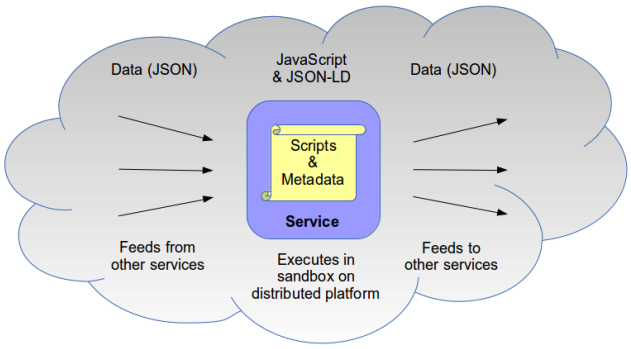

Figure 1: A Web of Value-Added Services

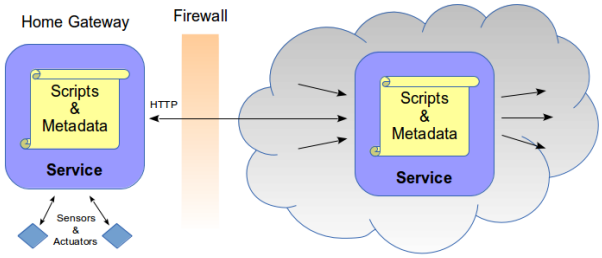

Figure 2: A Web of Services coupled to the IoT

Service Clouds

A service can be defined by scripts for the behaviour together with metadata that includes details on the developer, the owner, the API exposed by the service, and access control policies.

Figure 1 illustrates the idea of a service where the behaviour is defined in JavaScript. The metadata is defined in JSON-LD [19]. The service consumes and produces data expressed in JSON [18]. The service is executed in a security sandbox by a cloud-based platform.

Figure 2 illustrates how a cloud based service can build upon a service running on a server behind a firewall. In this case, the server is a home gateway and runs local scripts that interface to sensors and actuators in the home. The firewall blocks incoming connections, so the scripts running in the home gateway make client requests to URIs exposed by by the cloud-based platform. This is fine for delivering data or notifications into the cloud, but awkward when the service running in the cloud wants to invoke a method exposed by the service running in the home gateway. This doesn't fit naturally into the REST framework, and motivates a layered abstraction architecture that hides the complexity.

One approach involves polling the server for notifications and method invocations. Additional HTTP requests are needed to deliver method responses. This can be made more efficient if the server maintains a queue so that multiple notifications and method invocations can be delivered in a single HTTP response. If the queue is empty the server could keep the connection open in the expectation of a notification or method invocation.

A closely related approach uses long lived HTTP connections [21], where the server pushes a stream of messages as the content of an extended HTTP response. The client can stream notifications, method invocations and method responses as the content of the

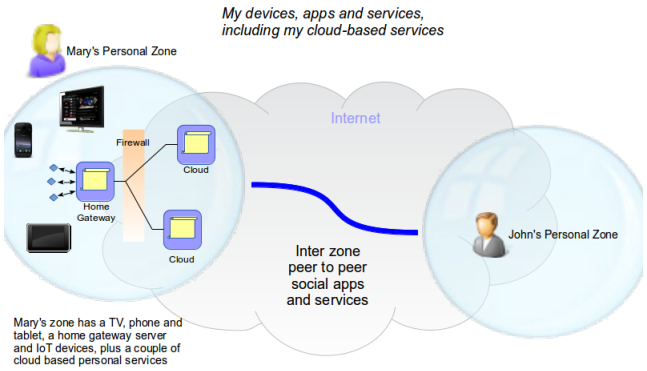

Figure 3: Personal Zones

HTTP POST request. The client-side script library that wraps this needs to be able to recover from inevitable closed connections.

A cleaner solution is for the script in the home gateway to establish a Web Socket [22] connection to the cloud and use JSON-RPC [20] as a basis for remote method invocation. Web Sockets provides a flexible solution for asynchronous bidirectional messaging over a TCP connection, and like HTTP can be used with transport layer security (TLS) to hide communications from scrutiny by third parties. JSON-RPC defines conventions for expressing method invocations, method responses and notifications as JSON messages. A lightweight scripting library can be used at both end points of the connection to map APIs to the corresponding message handling.

Another solution is for the home gateway to expose an HTTP server outside of the firewall so that cloud based services can directly connect to the gateway as needed. This could involve configuring port forwarding on the firewall to pass TCP connections requests to the home gateway. With current devices, this is cumbersome for end users to set up. Alternatively, the home gateway could integrate the function of the firewall, e.g. as a new class of device that acts as both a WiFi router, firewall and gateway.

Further complications arise when services behind different firewalls need to communicate. This generally involves the need for a third party to either relay messages, or to help set up peer to peer connections between a pair of services. W3C's Web RTC Working Group [23] is developing a standard means for scripts to set up such connections using NAT-traversal technologies such as ICE, STUN, and TURN.

A different approach has been studied by the European project named Webinos [17]. The starting point is the observation that increasingly people have many different personal devices from different vendors. A unified approach is needed to help people manage their personal devices, apps and services. The proposed solution is the idea of a *Personal Zone*. Each of your devices, apps and services are enrolled as part of this zone, which safeguards security, and also exposes zone APIs for inter-zone applications between different people[1]. The external interfaces for the Personal Zone are provided on servers that are directly accessible on the Internet, and as such exist outside of the firewall. This is illustrated in Figure 3.

Figure 4 illustrates how independent service clouds could work together based upon open standards as a basis for connecting services hosted by different clouds.

[1]Webinos implements personal zones with a Node.js based proxy server in each device. This exposes a rich suite of intra-zone APIs to trusted web applications, and facilitates cross device applications.

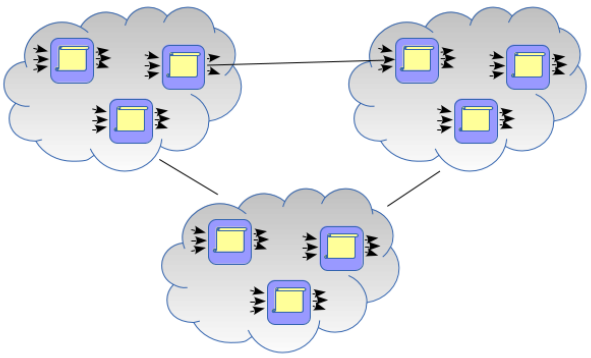

Figure 4: A Federation of Service Clouds

Service Platform

Given the complexity of communicating with services behind firewalls, it makes sense for the service platform to hide the implementation details from service scripts, and to support a variety of solutions as described in the previous section.

An analogy can be made between services and software packages on operating systems like Linux. Each package has a name and version number, and details of the names and version numbers for the packages it depends upon. This allows the package manager to identify and load the dependent packages when the user asks for a given package to be installed.

A similar approach can be used for the Web of Services. Each service needs to provide a name and version number, together with the details for the services it depends upon. This is expressed as part of the service's metadata. Rather than binding services by their name, a more general solution is to bind them based upon their type. This is a better fit for an open market of services, where competition is an integral aspect of the process of fulfilling evolving market needs.

What are the requirements for expressing the service type? It seems reasonable to try to make use of an existing interface definition language. This needs to support named interfaces and preferably some form of interface version numbers. The interface name needs to signify a functional behavior and semantics. A temperature sensor could report the temperature as a number, but we need to know that the corresponding interface is for a temperature in say, the Celcius scale. We could just call the interface *Celsius*, but this doesn't allow us to compare the semantics of different interfaces. First of all, we need globally unique names to avoid the risk of people independently using the same name for different things. Secondly, we need a framework for rich descriptions of services, including the means to distinguish different versions of a service. The Semantic Web [13] provides a solution to these requirements.

A popupular interface definition language for defining Web standards is the *Web Interface Definition Language* (Web IDL [24]). This currently lacks support for globally unique identifiers for interface names, however, this could be added with a minor change to the syntax. The basic syntax for Web IDL looks like:

```
[extended-attributes]
interface identifier {
  member definitions
};
```

The Web IDL specification suggests the use of extended attributes for language specific extensions, but this is aimed at the needs of the programming languages the IDL will be compiled into. A better approach might be to introduce a namespace prefix for identifiers and a means to bind these prefixes to URIs. This is possible as Web IDL grammar excludes the colon character from appearing within identifiers. Here is an example which assumes a hypothetical namespace for physical units:

```
@prefix phys http://example.com/ns/physical-units#

[extended-attributes]
interface phys:Celsius {
  attribute float temperature;
};
```

The above example declares Celsius as part of a given namespace. The namespace prefix syntax would also be usuable with interface names for attribute types, other than the types predefined in the Web IDL specification. There needs to be a way to retrieve external definitions of interfaces and Web IDL currently lacks an import statement. More far reaching extensions would address the ability to annotate interfaces in a variety of ways. You might, for example, want to indicate that a valid temperature reading is within the range from -40 to 100. String values could be constrained by regular expressions. Validity constraints could span multiple attributes.

Web IDL is just one example of interface definition languages. Another is the Web Application Description Language (WADL [25]), which uses XML for describing REST-based web applications. For the reasons stated earlier in this paper, it seems preferable to express service APIs at a level decoupled from the underlying use of HTTP, so WADL is unlikely to be a good solution for the requirements for a cloud services platform. The Web Services Description Language (WSDL [26]) is another XML based IDL language and targets interfaces based upon the Simple Object Access Protocol (SOAP).

The service platform executes scripts in a sand box that hides the details of how APIs for dependent services are mapped into the underlying transport protocols. This makes it easy to scale for increased demand by replicating services across cloud provisioned hardware. The platform needs to take care of service specific storage requirements, and synchronization across different instances of the same service. This is analogous to the requirements for client-side storage for web applications. In the Compose project, we are planning on using a NoSQL database for data represented in JSON, together with the SPARQL query language, [28] for data and meta data expressed as triples. JSONiq [29] could be useful as a query language for large amounts of JSON data.

Privacy and Security

Privacy is a very important topic for the Web of Things. Transport layer security (TLS) can be used to hide communications from third parties. Service owners can set policies for who can access the APIs exposed by the services they own. This is based upon a uniform treatment of identities together with strong authentication.

In the earlier example, Mary can set an access control policy that limits who can access her webcam. The starting point is to secure the connection between her webcam and the cloud based service she wants to use. This involves mutual authentication between the service in her home, and the one in the cloud. This can be based upon a secure exchange of public keys when the two services are bound together.

If Mary wants to provide her friend John access, she now needs to learn the public key that his software will used to identify him. There are a number of ways in which that could happen. She could

ask a mutually trusted service, e.g. a social network in which they are both registered as friends. If she knows John's email address, it may be possible to use the Web Finger framework, or John could pass her his public key using an out of band mechanism, e.g. as a QRCode on a business card, or through an NFC based exchange between their smart phones.

People may want to set data usage policies for data derived from their services. This may take the form of restrictions on the purposes that data can be used for, how long it can be retained, and who it can be shared with. Such policies could be attached to the data in some way that is preserved as the data flows through a web of services. Access control could also be sensitive to the context of use, for instance, I might grant you detailed access to my location only during my working hours. There is a need for exceptions in emergencies, so that people can be rescued and brought to safety.

One framework for handling access control decisions is XACML [30]. This is an XML format for access control rules together with a processing model that separates policy decisions from policy enforcement. According to Andras Cser at Forester Research, XACML is fading away [31]. This is in contrast to OAuth [32] which is steadily gaining in popularity. OAuth enables clients to access server resources on behalf of a resource owner, and was introduced to allow end-users to authorize a website to access their resources on a separate website without needing to disclose the credentials they use to authenticate to that site.

It is likely that the Web of Things will motivate the development of an approach that is designed to work with a web of services. It should be simple to make the service public, private to the owner, or accessible to closed set of people. This could involve named access control lists that are maintained by the owner or by a trusted third party. A further possibility would be to use public key certificates as an alternative to explicit access control lists – the certificate attests that someone is in a given set. A simple expression language could be used to attach conditions acting over an extensible set of properties, e.g. the time of day.

Privacy is dependent on the security of the platform and this is best done in depth so that attackers have to breach multiple layers of security to achieve their objectives. The scripting environment should preclude the use of insecure language features, e.g. JavaScript's eval. Services could specify what capabilities they need in advance, as in the Android operating system. Robust execution of scripts is likely to depend on valid data being passed to them. This is why it is important to consider how to include validity constraints as part of the service API definition language, and for the service platform to apply them and signal errors to the service scripts.

The platform could monitor services to detect irregular patterns of behaviour. Specialized services could be used to examine newly installed or upgraded services for signatures of malware in much the same way that we are used to with the Microsoft Windows and Android operating systems.

The Compose project is also looking at the potential for tracking provenance of data for controlling access based upon people's trust relationships and the need to preserve confidentiality agreements. If company A and company B are direct competitors, then leaking information from company A to company B will give the latter a potential commercial advantage. If you have a consultancy agreement with company A, and I have a consultancy agreement with company B, then I should not be given access to services you provide which are based upon confidential data from company A, even if you and I work for the same consultancy company, as otherwise I will be able to share this data with company B to their commercial advantage.

Reasoning about provenance requires some form of machine interpretable descriptions about how the data exposed by a service is related to the services it depends upon. There is also a need for descriptions relating to the confidentiality of the associated data sources, and the relationships between the people involved. This points to the utility of rich descriptions and a framework for reasoning over them.

Apps and Services

It is worth distinguishing between apps and services, where apps include a user interface that runs in a web browser, whereas services just expose an API. Web page scripts can access this API directly using a library that maps the API to the corresponding messages over HTTP or Web Sockets. This library can be generated automatically from the service's metadata. Packaged services can include apps and libraries for use with that service. This could be a complete web page, or a module for inclusion as part of a web page. Such modules could be based upon Web Components [33]. Other possibilities include scripts that dynamically generate the markup as part of a web page, or even PHP modules for server side inclusion.

Platform Tools and APIs

The service platform needs to provide a means for end users and developers to interact with the platform. The starting point is to register an account with the service cloud. End users could then use web applications provided by the service cloud to search for services, to read reviews, to add comments and ratings, to purchase and install services, and to bind them to match their specific needs.

Developers will want tools to create and maintain the services they develop. These could run within web browsers using syntax colouring for document editors, and graphical tools for constructing a web of services. *Yahoo! Pipes* provides a precedent [34]. You would need a tool for searching and browsing through services, and a graphical means for connecting them together. There is also a need for analytic tools for monitoring the use of the services you provide.

A Worked Example

This section provides a sketch of a service for a networked security camera, e.g. as shown in Figure 5. These are available at low cost and feature motion detection, and pan, tilt and zoom control. Such devices commonly include an HTTP server and can be controlled via a web page accessed from this server. The number of simultaneous users is limited to 5 visitors. The camera can be panned through 355 degrees and tilted through 120 degrees.

A simplified API would allow for panning and tilting the camera, viewing the video stream, and receiving notifications when a moving object is detected in the field of view. Here is the WebIDL:

```
enum Quality { "high", "medium", "low" };

interface WebCam {
  attribute float azimuth;
  attribute float elevation;
  attribute Quality resolution;
  readonly attribute DOMString videoURL;
  callback motionDetectedCallback = void ();
};
```

This definition lacks validity constraints on values, e.g. limiting elevation to be between plus and minus 60 degrees. The camera can be controlled from JavaScript by assigning values for the azimuth and elevation. These assignments map into HTTP POST requests, where the request payload is expressed as JSON, e.g.

Figure 5: Remotely controllable camera from Shenzhen EasyN Technology Company.

```
POST settings HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64)
Keep-Alive: 300
Connection: keep-alive
Content-Type: application/json
Content-Length: 49

{"azimuth":180,"elevation":0,"resolution":"high"}
```

The videoURL is a readonly property that can be used to display the video, e.g. in an HTML5 video element. The motionDetectedCallback property can be set to a method that you want called when the camera detects something moving in its field of view. The API could be easily extended to allow you to remotely control the brightness, contrast, saturation, and hue. Further possibilities include saving video to the cloud and logging motion detection events.

The service description is expressed in JSON. The following is over simplistic, but gives a basic idea of what is involved:

```
{
  "Developer" : "Shenzhen EasyN Technology Company",
  "Version" : "1.0.0",
  "Owner" : "Mary Smith",
  "API" : "webcam.idl",
  "Behavior" : "behavior.js"
}
```

The names of the developer and owner aren't sufficient as unique identifiers. One possibility is to use email addresses as these would work across service clouds as compared to account names that are internal to a given cloud. The API field links to the IDL for the API the service exposes. The Behavior field links to the script that implements the API. This service doesn't depend on other services. If it did, there would need to be a Dependency field with a list of dependencies.

When Mary sets up the service, it needs to be bound to her webcam. This could be based upon a static IP address and port, or using a domain name and relying on dynamic DNS to work around dynamically assigned external IP addresses. For services that build upon other services, the binding process likewise needs to link to specific service instances. The service platform needs to provide an API for the binding process.

This example needs extension to cover access control preferences. By default, the service is only accessible by its owner. If Mary wants to make it available to anyone, she needs to include the field *Access* with the value *public*. She could instead list the people she wants to give access to. The IDL example likewise needs extension to support namespaces for globally unique vocabularies, as explained earlier in this paper. Further work is needed to explore the potential for using JSON-LD for semantically rich descriptions of services that complement the API definition in an interface definition language. One possibility would be to use the ontology developed by the W3C Semantic Sensor Network Incubator Group [35].

4. CONCLUSION

This paper outlines an architecture for the Web of Things that blends the Internet of Things and the Semantic Web for distributed service platforms as a basis for open markets of services. The platform simplifies the effort needed for creating services by hiding details best handled at a different level of abstraction. This applies in particular to the use of HTTP for passing data between services.

A range of programming languages will be needed, e.g. simple expressions for data conversions, scripting languages for flexibility, rule-based languages for logical inferences, and specialized languages for exploiting GPUs for efficient operations on numeric data. JavaScript is expected to be a popular choice given its existing use in web applications. It makes sense for the service platform to re-use APIs that developers are familiar with from client-side scripts.

5. ACKNOWLEDGMENTS

This work was supported by the European Commission through funding for the FP7 Compose project (FP7-317862) [16].

6. REFERENCES

[1] Uniform Resource Identifiers.
RFC3986, January 2005, URI Syntax.
http://www.rfc-editor.org/rfc/rfc3986.txt
RFC3305, August 2002, Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations. http://tools.ietf.org/html/rfc3305

[2] Hypertext transfer protocol (HTTP 1.1). RFC2616, June 1999. http://www.ietf.org/rfc/rfc2616.txt.

[3] Hypertext Markup Language (HTML5) http://www.w3.org/TR/html5.

[4] Cascading Style Sheets (CSS) http://www.w3.org/Style/CSS/Overview.en.html.

[5] JavaScript – the familiar name for the ECMAScript programming language http://www.ecmascript.org/docs.php.

[6] W3C Web Architecture. W3C Recommendation, 15 December 2004. http://www.w3.org/TR/webarch/.

[7] Companies offering GPU cloud computing services. http://www.nvidia.com/object/gpu-cloud-computing-services.html.

[8] GPU powered neural networks for image recognition http://www.pcworld.com/article/2042339/nvidias-gpu-neural-network-tops-google.html.

[9] Scale Free Networks Wikipedia article. http://en.wikipedia.org/wiki/Scale-free_network

[10] Small-World Network Wikipedia article. http://en.wikipedia.org/wiki/Small-world_network

[11] According to Kevin Ashton, he coined the term "Internet of Things" in 1999 in a presentation to P&G, see his article on the Internet of Things dated 22 June 2009. http://www.rfidjournal.com/articles/view?4986KevinAshton,22June2009.

[12] Invited presentation to students at the University of the West of England, 26 September 2007, http://www.w3.org/2007/Talks/0926-dsr-WDC/slides.pdf

[13] The Semantic Web – a framework for machine interpretable data and metadata based upon universal identifiers, and <*subject, predicate, object*> triples, http://www.w3.org/standards/semanticweb/.

[14] ABB, Bosch, Cisco & LG: Open Standard for Smart Homes, 1 November 2013, http://www.bosch.lt/en/lt/newsroom_17/news_16/news-detail-page_38208.php.

[15] Cross Origin Resource Sharing (CORS), http://www.w3.org/TR/cors/.

[16] EU FP7 Compose Project, http://www.compose-project.eu/.

[17] EU FP7 Webinos Project, http://www.webinos.org/.

[18] JSON – JavaScript Object Notation http://json-rpc.org/

[19] JSON-LD – Linked Data in JavaScript Object Notation http://www.w3.org/TR/json-ld/, and http://json-ld.org/

[20] JSON-RPC – Remote procedure calls using JSON messages http://json-rpc.org/

[21] Long-Lived HTTP Streams http://ajaxpatterns.org/HTTP_Streaming

[22] WebSockets – Asynchronous messaging protocol and API http://tools.ietf.org/html/rfc6455 and http://www.w3.org/TR/websockets/

[23] Web Real-Time Commmunications Working Group http://www.w3.org/2011/04/webrtc/

[24] Web Interface Definition Language (Web IDL) http://www.w3.org/TR/WebIDL/

[25] Web Application Description Language (WADL) http://en.wikipedia.org/wiki/Web_Application_Description_Language

[26] Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl

[27] Simple Object Access Protocol (SOAP) 1.2 http://www.w3.org/TR/soap12-part1/

[28] SPARQL Query Language for RDF, v1.1 http://www.w3.org/TR/sparql11-overview/

[29] The JSON Query Language (JSONiq) http://www.jsoniq.org/

[30] Introduction to XACML http://en.wikipedia.org/wiki/XACML

[31] XACML is dead, Andras Cser, 7 May 2013 http://blogs.forrester.com/andras_cser/13-05-07-xacml_is_dead

[32] Introduction to OAuth http://en.wikipedia.org/wiki/OAuth

[33] Introduction to Web Components http://www.w3.org/TR/components-intro/

[34] Yahoo! Pipes Tutorial http://pipes.tigit.co.uk/

[35] Semantic Sensor Network Ontology http://www.w3.org/2005/Incubator/ssn/ssnx/ssn