Hypermedia APIs for Sensor Data

A pragmatic approach to the Web of Things

Spencer Russell
sfr@media.mit.edu

Joseph A. Paradiso
joep@media.mit.edu

Responsive Environments Group
MIT Media Lab
Massachusetts Institute of Technology
Cambridge, MA, USA

ABSTRACT

As our world becomes more instrumented, sensors are appearing in our homes, cars, and on our bodies [12]. These sensors are connected to a diverse set of systems and protocols driven by cost, power, bandwidth, and more. Despite this heterogeneous infrastructure, we need to be able to build applications that use that data, and the most value comes from integrating these disparate sources together. Infrastructure for the Internet of Things (including not just consumer products but sensors and actuators of all kinds) is becoming more commonplace, but we need an application layer to enable interoperability and create a *Web* of Things. Here we introduce a pragmatic, hypermedia approach to the Web of Things, integrating HTTP request/response interactions with realtime streaming using HTML5 WebSockets. We will discuss how our approach enables client/server interactions that are both evolvable by the server and discoverable by the client. Rather than attempt to define yet another competing standard, we incorporate a collection of complementary standards already in use. We will also describe our implementation of these concepts in ChainAPI, a sensor data server in use by a variety of projects within our research group. We will describe one of several end-to-end applications as a successful case study.

Keywords

Hypermedia, Internet of Things, Linked Data, RESTful Web Services, Semantic Web, Sensors

1. INTRODUCTION

It is becoming apparent that in addition to a transport layer that enables the Internet of Things, it is important to develop an application layer to provide wide-spread interoperability and a consistent interface to Internet-connected devices. While there are many efforts to develop new standards and protocols such as AllJoyn[1] and MQTT[2], other projects [20] seek to use existing application-level Web standards such as HTTP to provide an interface that is more familiar to developers, and also that can take advantage of tooling and infrastructure already in place for the World Wide Web. These efforts are often dubbed the Web of Things, which reflects the relationships to existing Web standards and also the way in which the World Wide Web is built on top of the Internet.

We can see some of the potential for Web-accessible sensor data in the growth of services like Xively[3]. However, we believe that the Web of Things should, like the Internet itself, be open and decentralized. Enabling interoperability between millions of data providers and consumers means focusing on the interactions between those actors, not building centrally-controlled services.

In previous work [13][18] we have built frameworks to collect and process sensor data from a variety of sources, as well as applications to visualize and experience those data. Through these prototypes we identified several common use cases and access patterns, as well as shared functionality that would be better served by a common infrastructure.

Many of the main impediments to adoption of IoT standards are social rather then technological. Often solutions require developers to take on too much simultaneous complexity to get started. The Semantic Web gives possible directions to encourage interoperability, but many of those systems are very complex, with highly-sophisticated data models. Ensuring compatibility with existing upper ontologies [9][7][14] improves interoperability, but application developers are often unwilling to adopt the accompanying complexity. Providing simple, familiar interfaces to sensor data lowers the barriers to entry, allowing developers to build sophisticated applications without deep knowledge of the underlying sensor architecture, enabling smarter, more efficient IoT systems [2].

To address these issues we have developed ChainAPI, a set of Web service design principles for the Web of Things.

[1]https://www.alljoyn.org/
[2]http://mqtt.org/
[3]https://xively.com/

MOBIQUITOUS 2014, December 02-05, London, Great Britain

DOI 10.4108/icst.mobiquitous.2014.258072

ChainAPI services interoperate with existing infrastructure, and also allow developers to take advantage of semantic relations and formal ontologies as they become useful, rather than forcing the developer to confront them all at once. We are working with developers outside our research group to get insight and feedback into the barriers to adoption and also which features are necessary to cover real-world use cases.

2. HYPERMEDIA WEB SERVICES

The seminal work in Hypermedia Web Services is Roy Fielding's PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures" [15]. Fielding codified much of the design that had gone into the World Wide Web into an architectural framework he called Representational State Transfer, or REST. He lists the main requirements that drove the design of the World Wide Web:

1. Low barrier to entry
2. Extensibility
3. Distributed hypermedia
4. Internet-scale

The Internet of Things certainly must be extensible and Internet-scale. This work focuses on exploring the benefits of lowering barriers to entry and hypermedia. While reducing complexity and providing easy entry points for new developers are obviously good goals in isolation, there is often a trade-off between design complexity and the expressive power and generality of a system. The World Wide Web gives a compelling model of a system that has proved itself to be both accessible (driving wide and rapid adoption) and extensible. Guinard, Trifa, and Wilde give a good introduction to the application of RESTful design to the Web of Things [20]. In particular they list five constraints of REST architectures:

1. Resource Identification
2. Uniform Interface
3. Self-Describing Messages
4. Hypermedia Driving Application State
5. Stateless Interactions

They go on to describe how these constraints are well suited to IoT applications. Building on that work and our previous experience we developed seven design principles to guide our system design, architecture, and implementation.

1. Assume a plurality of low-level device protocols and abstract them from clients
2. Support a layered architecture allowing intermediaries to handle services such as caching, authentication, and encryption
3. Use hyperlinks to present resource relations and client affordances
4. Support interoperability via shared vocabularies identified with URIs
5. Developers should be able to build clients without specialized tooling or libraries, using only familiar standards (HTTP, JSON, and WebSockets)
6. Provide semantic relationships without requiring full Semantic Web buy-in from developers
7. Provide a mechanism for clients to subscribe to push updates

3. RELATED WORK

There are a wide variety of projects in the academic and private sectors, and in fact the abundance of fragmented systems and protocols is often cited as one of the main challenges in IoT. To help make sense of this landscape we will categorize some of these related projects into 3 groups: data interchange formats, protocols, and platforms. In this categorization we consider data interchange formats to be portable specifications intended to package data for sharing between heterogeneous systems. Protocols specify more complex interactions between entities, such as acknowledgments, authentication, and handshaking. A protocol may specify the data interchange format, or may leave the payload format open for arbitrary data (as HTTP does). Platforms are specific instantiations and implementations in hardware or software.

3.1 Data Interchange Formats

On the Web, JSON and XML are both extremely common and often used as a basis for more detailed formats. This paper will not contribute further to the JSON/XML discussion, except to point to the well-reasoned description by Lanthaler [23] which aligns with the authors preference for JSON.

The Sensor Modeling Language (SensorML) from the Open Geospatial Consortium (OGC) is a data model with an XML representation that can describe sensor metadata. Though widely used and referenced in the literature, SensorML is burdened by interoperability with general upper ontologies and does not seem suitable for lightweight web services.

While not sensor-specific, the Hypermedia Application Language (HAL) [21] provides a data model with both XML and JSON formats that supports hypermedia i.e. links. HAL has a focus on simplicity, but has the ability to use URIs as link relations, which enables a global vocabulary that can be shared between applications. A downside to HAL is that relation URIs can only be used for link relations, not resource attributes. JSON-LD [31] is another promising contender in the JSON hypermedia space, and unifies the descriptions of links and attributes. It has recently been adopted by Google for embedding semantic data into emails and search results [30]. JSON-LD can also be mapped to RDF, for interoperability with existing Semantic Web tools.

3.2 Protocols

Much of the work in IoT research and development has been at the protocol level. CoAP [29] and MQTT [24] both target protocols that can be used end-to-end from embedded devices to clients. CoAP is inspired by REST and HTTP but designed from scratch by the IETF [22], while MQTT is modeled on a publish/subscribe architecture. XMPP began as a chat protocol, but because it supports extensions it has been used for a wide variety of use cases, including IoT. Any of these three would require adapting into Web standard protocols such as HTTP, so while they could be plugged into a system meeting our design principles, we will consider them out of scope for this work. AllJoyn is a large and sophisticated IoT solution started by Qualcomm that is now under the AllSeen Alliance. AllJoyn does not fit into our design principles of using existing protocols when possible and avoiding the need for extra tooling.

3.3 Platforms

There are many commercial and academic offerings to manage sensor data. One of the most well-known is Xively (formally Cosm, formally Patchube). Xively supports access via REST, Unix sockets, WebSockets, and MQTT. However, it is a closed and monolithic system. Their integration of request/response API and push API was influential in our design, though they are not leveraging hypermedia, and rely on out-of-band URLs hard-coded into the client.

In 2014 SparkFun announced data.sparkfun.com, which is a very simple sensor feed publishing platform. It is open-source, but the lack of a mechanism to link installations together prevents it from becoming a true distributed system. Also, While it is very simple and accessible to web-savvy developers, it doesn't provide the necessary tools to build more complex systems.

Microsoft Research has introduced HomeOS [11], which abstracts over a set of existing home-automation protocols to provide a PC-like interface, but it is very home-focused and also does not provide a web-centric API to client developers.

SPITFIRE [25] is a very promising step towards the Semantic Web of Things, and motivates the benefits of semantic sensor data well. However, they do not provide a hypermedia API and instead rely on the use of Semantic Web tools such as SPARQL. OpenIoT [1] is a similar service based on Semantic Web technologies and conforming to standards such as the W3C Semantic Sensor Networks (SSN) specifications. Again however, the system is large and complex, and relies on tools and specialized software to interact with the service.

4. USING EXISTING STANDARDS

Wherever possible we have relied on existing standards and protocols rather than reinventing our own. For instance, to support hypermedia in our responses we are using HAL. HAL has a large number of client libraries and was recently adopted by Amazon for their AppStream API. HAL can be rendered in JSON with a media type of `application/hal+json` or XML (using `application/hal+xml`). In this case we have chosen to focus on the JSON variant. JSON-LD would also be a suitable representation, and they can co-exist using the HTTP Accept header to let the client tell the server what representation it wants. We use the IANA-standardized link relations `edit-form` and `create-form` to indicate link relations that can be used for editing and creating resources, and JSON-Schema [33] for representing the expected format. For our realtime API we are using WebSockets, which is a simple and low-overhead protocol with wide and growing adoption. Where possible we use existing vocabularies, such as the QUDT[4] ontology for unit names.

5. LAYERED ARCHITECTURE

Work is under way by multiple groups to adapt the TCP/IP Stack to be more suitable for low-power, resource-constrained devices [3]. Though this is a reasonable proposition and would provide a suitable transport protocol for communication, it leaves open many questions that are important for secure and reliable communication over the open Internet. Even if the devices themselves speak IP (whether using WiFi, 6LoWPAN, etc.) there will still likely be a role for a bridge or gateway node that can handle encryption, authentication, discovery mechanisms, and other functionality necessary to communicate and interoperate with the larger Internet.

One of the benefits of the layered gateway approach is that we can take advantage of existing HTTP Caching and Proxy infrastructure. It is a common pattern in modern web development to have application web server processes handling application logic, and to place a front-end HTTP server such as Nginx or Apache as a reverse proxy or gateway. In this configuration, the proxy is responsible for handling SSL encryption, defending against DDOS attacks, and in general provides a front line of defense to the open Internet. The application server processes (e.g. Node.js or Gunicorn) thus operate in a safer environment and focus on handling application logic. This architecture maps directly to IoT applications, where lightweight devices can serve HTTP or function as gateways to Wireless Sensor Networks (WSNs) from behind heavier-duty proxies.

The presence or absence of a gateway has been used in Web of Things literature to divide direct from indirect integration [32]. In direct integration, the devices themselves are capable of serving requests directly from clients. In indirect integration there is a gateway or bridge that serves the requests, and translates them to a protocol that the (presumably lower-powered) nodes can understand. The assumption is often that the main function of the gateway is to translate between HTTP and a lower-power protocol such as ZigBee, 6LoWPan, etc. An unexplored middle ground is for end devices to function as simple HTTP servers that are proxied behind more heavy-duty ones. For example, a gateway could handle SSL and authentication, but then forward requests to the simpler devices for handling application logic. This layered architecture and allows the simpler devices to sit behind a protected firewall in a safe zone. By building our simpler servers on HTTP we can use industry-standard and field-tested front-end proxy servers instead of special purpose IoT gateways.

Figure 1 shows a possible configuration of clients, proxies, and sensors. Here there are a combination of traditional low-power sensors in a WSN, as well as a set of sensors that can serve HTTP requests, but are assumed to be relatively resource-constrained i.e. they are not equipped to handle large amounts of traffic, and don't support advanced features such as SSL. When one of the clients requests data (e.g. the most recent temperature reading) from one of the sensor nodes, it goes through a caching proxy. If the proxy does not have a recent response, it will pass the request to the sensor. The sensor will send the response with the data, along with a standard HTTP Cache-Control header. The sensor would likely set the cache lifetime to persist until its next scheduled measurement. Subsequent requests from other clients for the same data will be handled directly by the Caching proxy, saving both power and bandwidth of the sensor. This is particularly valuable when there are many clients requesting the same data.

[4]http://www.qudt.org/

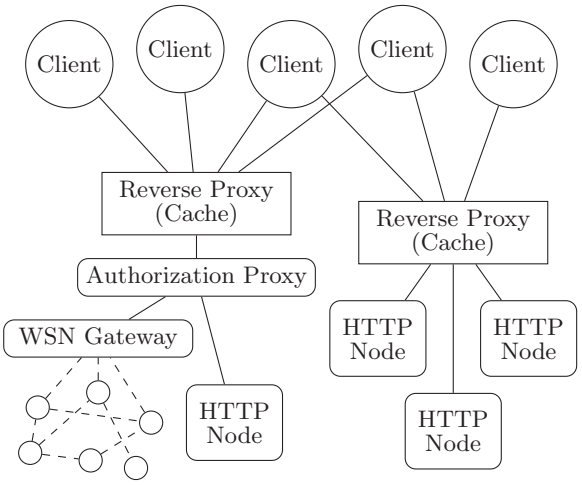

Figure 1: Layered Architecture

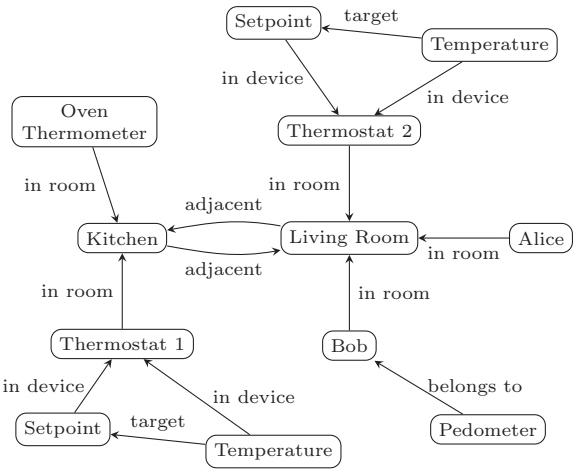

Figure 2: Linked Resources

The trade-off for this extra efficiency is additional latency between the time a physical property is measured and the time that it gets to the client. Because our system provides both a request/response and a streaming interface, we assume that clients seeking the lowest latency data will simply connect to the realtime stream.

The standard HTTP methods (`GET`, `POST`, etc.) have well-defined semantics [16]. For instance, a `GET` request should have no side effects in the server, and a `PUT` request should be idempotent (making the request more than once has the same effect as making it once and the request can be safely repeated). Using these standard methods and adhering to the defined semantics allows intermediate servers to behave more intelligently and route traffic more efficiently. One notable example is the caching proxy. There exist many widely-used proxies such as Varnish[5], Nginx[6], and Squid[7] that can take advantage of the HTTP method along with standardized cache lifetime HTTP headers to intercept the request and serve a cached response from a previous request. This reduces the load on the application server, which might otherwise have needed to access a database or do other expensive calculations to re-build the response. In the case of sensors the caching proxy could be a power-saving measure, as the proxy could handle client requests without communicating with the sensor.

6. LINK RELATIONS

Hyperlinks provide a mechanism for the server to present affordances [17] to the client. These affordances could represent actions that the client has available to them, or simply related resources. For example, Figure 2 shows a set of resources and relations in a home. We see that Alice and Bob are both in the Living Room, where there is also a thermostat. There are two tracked values in the thermostat: the setpoint and the temperature, and we can see that the setpoint can be considered a target value for the temperature. In a hypermedia system each of these entities can be represented by a resource with a unique URI, and each of these relations (adjacent, in device, etc.) can be represented by a hyperlink. This is also similar to the subject-verb-object triple concept from the Semantic Web [5]. Linked resources could also be virtual sensors such as an anomaly detection algorithm that uses raw sensor data as input. Actions such as editing the current resource or sending a command can also be represented as links, and the presence or absence of those links is a way for the server to communicate to the client what they are allowed to do. Listing 1 has a typical resource representation. In hal+json links are contained in a reserved `_links` property of the JSON payload. The `_links` property is a dictionary, keyed on link relation names (commonly referred to as "rels").

The rel itself actually serve as a link to the human-readable documentation that describes what that relationship actually means. This rel URI should also be used by clients as a unique, persistent identifier. This gives flexibility to server implementers as they can add new relation types, or even new versions of existing relation types, to existing resources without breaking older clients. As long as the new rels have unique URIs, old clients will simply ignore them. It also provides a mechanism for creating a shared vocabulary of relation types by referencing rels from a shared repository.

Using URIs as relation names has the benefit of providing a stable and unique identifier for relation names, but using a full URI as a JSON dictionary key is cumbersome and duplicates lots of data in a typically payload. To alleviate this issue hal+json supports Compact URIs or CURIEs [6], which are similar in functionality to an XML namespace. In the context of hal+json CURIEs are simply a URI template that can be used for each rel that references it. For example, in Listing 1 the `ch` CURIE has a templated URI of `/rels/{rel}`, so the rel `ch:device` becomes `/rels/device`. This substitution is known as CURIE Expansion.

Actions such as posting new sensor data or adding a new sensor require the client to send data to the server. The client can send data to the server via `POST` requests with data encoded in the body of the request with JSON. The server can provide the client with the expected format of the data using JSON-Schema [33]. A `GET` request to the given

[5]https://www.varnish-cache.org/
[6]http://nginx.org/
[7]http://www.squid-cache.org/

relation will generate a response with the expected schema, and a `POST` will edit or create the resource, depending on the relation.

You can also see in Listing 1 that there is both an `editForm` and `edit-form` relation. Notice that the `editForm` link has a `deprecation` property that is a URI. This is how relations can be gracefully deprecated or changed. In this case we renamed `editForm` to `edit-form` to match the IANA-registered relation name. Older clients with the old relation name will continue to function normally, and most HAL libraries are configured to log a warning if a deprecated link relation is used. The deprecation URI can be viewed for information about the deprecation. Developers creating new clients can clearly see which relations are deprecated and avoid using them. HAL allows server developers to decide on a deprecation lifetime policy appropriate for their application.

To save on communication overhead, HAL also supports embedded resources as a pre-caching strategy [21]. If there are related resources that are likely to be requested, they can be included in an `_embedded` property that is similar in structure to the `_links` property, except with full HAL resources included instead of links. When a client application accesses a related resource, most HAL client libraries are configured to first check to see if the resource is embedded before requesting the it over the network.

7. BRINGING POLL AND PUSH TOGETHER

In ChainAPI we integrated request/response interactions with realtime push updates. While many interactions map well to an HTTP API in which clients make a request and the server sends a response, it is very common in sensor data systems to want new sensor data as soon as it is available. Standard HTTP is not well-suited to this task, as the server cannot initiate communication with the client. While many workarounds such as long-polling are in common use, HTML5 WebSockets were introduced to provide a more standard mechanism.

Clients begin interacting with ChainAPI via the HTTP API, submitting HTTP requests for resources and receiving responses with representations of those resources in hal+json. In a response, the server may provide a realtime feed for related data simply by providing a `ch:websocketStream` link relation. For example, in Listing 1 the server is telling the client that there is a WebSocket stream for this sensor available at `ws://example.com/ws/sensor-274`. As new data from the sensor is available it will be published to all subscribed clients. Because there is a natural hierarchy in our data (many devices in a Site, usually several sensors in each Device), when clients subscribe to a resource they get updates for all resources below them in the hierarchy.

This is a good example of the benefits of hypermedia and linking, as the client does not need to be instructed ahead of time how to find a given realtime stream, it can just follow the link at run-time. This frees the client developer from needing to figure it out from documentation, and also allows the server developer to make changes as needed without breaking clients. For instance, the streams could be served

```
{
  "updated": "2014-04-01T02:34:21.676564+00:00",
  "dataType": "float",
  "metric": "sht_temperature",
  "value": 25.59,
  "_links": {
    "ch:dataHistory": {
      "href": "/sensordata/?sensor_id=274",
      "title": "Data"
    },
    "curies": [
      {
        "href": "/rels/{rel}",
        "name": "ch",
        "templated": true
      }
    ],
    "self": {
      "href": "/sensors/274"
    },
    "ch:device": {
      "href": "/devices/33",
      "title": "Office Thermostat"
    },
    "ch:websocketStream": {
      "href": "ws://example.com/ws/sensor-274",
      "title": "Websocket Stream"
    },
    "editForm": {
      "deprecation": "/rels/deprecation/editForm",
      "href": "/sensors/274/edit",
      "title": "Edit Sensor"
    },
    "edit-form": {
      "href": "/sensors/274/edit",
      "title": "Edit Sensor"
    }
  },
  "unit": "celsius"
}
```

Listing 1: hal+json representation of a sensor

from a totally separate server, or the HTTP server might want to pass additional context information to the streaming server through parameters in the URI query string.

Another option with some traction is MQTT [8], which is optimized for resource-constrained devices and includes useful features such as variable Quality of Service levels and the ability to register a message to be sent on disconnect, known as the "last will and testament." For our applications these have not been necessary and the simplicity of integrating WebSockets with standard HTTP has proved beneficial. If in the future we decide to support additional streaming protocols such as MQTT we can simply add new link relations to allow clients to connect via those protocols, analogous to the `ch:websocketStream` relation we currently use for the WebSocket stream. Clients that support the new stream type will be able to take advantage of it when the rel is

available, and older clients will simply ignore the new capability.

8. ENABLING SEARCH

In the early days of the World Wide Web, sites were primarily accessed directly via their URIs, or via links from other known sites. In the earliest days in fact, CERN had an alphabetized index of available web content [28]. As the web grew beyond what could be reasonably browsed and bookmarked, the problem of finding information on the Web changed. It was no longer enough for the content to be on the web, it also had to be discoverable in a sea of other pages. By 1994 search engines started to appear to allow users to find the information they wanted [28]. With search engines came the advent of crawlers that would index the web and collect the metadata into the engine's database.

As more sensors and sensor networks are added to the Web of Things, similar issues will arise. By building a network of interlinked sensor networks, our approach can serve as a substrate on which a similar ecosystem of crawlers can index the available data and metadata. Search engines have also been one of the spaces where semantic markup is beginning to be adopted and used, as the engines use embedded semantic information in pages to improve their results.

9. SUPPORTING RELATION ONTOLOGIES

One of the central issues in the IoT is simply the issue of uniquely identifying objects in the system [3]. RESTful design practices encourage HTTP URIs as globally unique identifiers. Providers can structure their URIs arbitrarily, for instance to represent natural hierarchy in the system. Link relations between objects not only represent identity, but also where the linked object can be found, without needing to first consult any sort of central registry. This architecture supports extremely loose coupling between related resources and services.

Additionally, HTTP has built-in mechanisms to handle renaming, as servers can respond with an HTTP Status 301 (Moved Permanently) to notify clients that the object can now be found at a new URI.

An ontology is a formal description of objects, classes, and concepts in a domain, as well as the relations between them [19]. In the Semantic Web context the entities in an ontology are typically defined as a set of widely-available URIs so that unrelated actors and systems can reference shared concepts [5]. In fact, it's this ontological common ground that enables the most powerful aspects of semantically-linked data. Client agents can take data from disparate sources and leverage the ontology to combine it meaningfully, for example combining temperature data from multiple sources, or automatically combining sensor data from different systems that it knows are nearby a particular latitude and longitude.

Implementing a fully Semantic Web-compatible system is not a priority of ChainAPI, but we recognize the benefits of shared ontologies to support cross-system compatibility. ChainAPI-based services can use a shared vocabulary of link relations to enable these types of use cases. In Listing 1, notice the `ch:dataHistory` link relation, which expands (via the CURIE) to the URI `/rels/dataHistory`. A client that sees and understands this relation knows that it will link to a collection of data from the history of this resource. A user-facing client might open the link in a separate window that can graph the data. A Machine-to-Machine (M2M) agent such as an indexing crawler might choose to ignore the link because it is only concerned with sensor metadata and not the actual measurements themselves. Perhaps it will link back to the source for the raw data, while storing the metadata in a more optimized index for the types of queries it will run.

In addition to acting as a universally-unique identifier, the rel name is also a URI that can be dereferenced by the client to get more information on the relation semantics. The human-readable documentation available at that URI is useful during development, but machine-readable documentation can also be made available, such as a JSON-Schema. This machine-readable information could be used by a hypermedia client to display more information about the relation to the user, or to decide whether or not to follow the relation in an M2M context.

Where possible we are using existing standard relation names. Some that apply to the wider web are defined by the IANA, such as `next`, `previous`, `edit-form`, etc. There are currently several available approaches to an ontology for sensor data [9][7][14] but they typically focus on interoperating with more universal upper ontologies and are too complex for use by general web developers. Other work on integrating various ontologies with other existing sensor description standards such as SensorML are promising [26], but finding the right balance between simplicity and semantic expressivity remains an open research area.

A shared vocabulary of relation types is one part of the semantic picture, but for a fully self-describing service it is also necessary for the client to access information about the resource formats themselves. The IANA defines the `profile` relation name that can be used to link individual resources to a shared type that can be used as a context for interpreting its attributes.

10. IMPLEMENTATION

To validate our design choices and experiment in a real-world environment, we have implemented a server-side web service, the ChainAPI server. We have also created several client applications to use the service in different ways. The software is released under the MIT license, and source code is available on GitHub[8].

10.1 Resources

As a hypermedia web service, ChainAPI provides clients with a number of resources and describes relations between them in the form of hyperlinks. The resource types we currently provide are:

Site A collection of Devices typically located within the same geographic area or building.

Device A physical device in an enclosure. This device could contain many sensors.

[8]https://github.com/ssfrr/chain-api

Sensor A single metric that is measured, such as temperature or humidity.

SensorData Raw data captured by the sensor.

10.2 Libraries and Tools

Our current implementation of the ChainAPI server is written in Python, and diagrammed in Figure 3. We use the Django[9] web framework for the request/response API and database interactions. For managing the WebSocket connections, we use a separate process built on the Flask[10] web framework. The two processes communicate with each other through a ZMQ[11] socket for event notification. We are currently using the PostgreSQL[12] relational database. Nginx acts as a reverse-proxy server to dispatch standard HTTP and also WebSocket connections to the application servers.

It is currently only possible to create or modify resources (including `POST`ing sensor data) through the HTTP interface, but in the future clients will be able to use WebSockets to reduce latency and overhead. When a client posts new data, for example a sensor posting a new temperature measurement, the HTTP server stores the data in the database and also sends an event notification over ZMQ to the WebSocket server, which in turn notifies any subscribed clients. The URI provided to the client in the WebSocket links provides all the information the WebSocket server needs to decide what data should be sent to the client. For example, in the sensor resource in Listing 1, the WebSocket URI is `ws://example.com/ws/sensor-274`, so when the client connects to the WebSocket server, it will send any events tagged `sensor-274`. This tagging mechanism is an implementation detail, but it demonstrates that when the clients treat link URIs as opaque, they can be used to pass data between server components without requiring the state to be maintained in the server, which is a core component of RESTful design.

While are certainly a long way from Internet-scale, we have a substantially larger installation then most comparable research systems. As of October 24, 2014 there are 487 devices in the system, which include 2230 separate sensors. We have collected over 239,000,000 sensor data measurements. These devices include custom hardware that communicates over 802.15.4, commercial thermostats on the MIT Media Lab's Building Management System, and the Soofa solar benches[13] deployed in Boston, which are `POST`ing directly to ChainAPI over a GSM modem.

10.3 HTML Interface

To assist developers building applications on top of ChainAPI, we have developed a human-facing interface[14] with HTML, CSS, and JavaScript that can be viewed through a standard web browser, as seen in Figure 4. Through this interface developers can familiarize themselves with the ChainAPI interface before they start their client application, or to evaluate

[9]https://www.djangoproject.com/
[10]http://flask.pocoo.org/
[11]http://zeromq.org/
[12]http://www.postgresql.org/
[13]http://www.soofa.co
[14]http://chain-api.media.mit.edu/

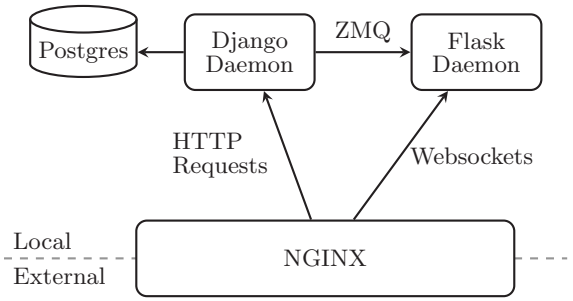

Figure 3: ChainAPI Server

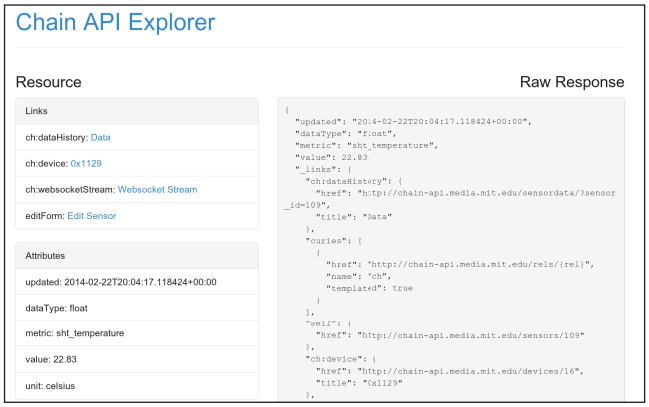

Figure 4: Chain API Explorer

ChainAPI for their use case. For each request, we display the raw JSON of the response on the right side. On the left we display a more user-friendly and interactive rendering of the raw data.

Our HTML interface has the following capabilities

1. Display resource attributes
2. Display links as clickable HTML links
3. Create and `POST` an HTML form from a JSON-Schema
4. Plot time-series data on a graph

Capabilities 1-3 are enough to enable a user to fully interact with the API without any specialized code on the client side. Plotting time-series data is a convenience to help visualize the data, but is not a core requirement of client libraries. As new features and capabilities are added to the server, they automatically become available in the client interface with no client-side code changes.

11. CASE STUDY: TIDMARSH LIVING OBSERVATORY

The Tidmarsh Living Observatory project[15] is a sensor deployment at a former cranberry bog in southern Massachusetts that is currently undergoing a restoration to a natural wetland. There are currently 65 sensors nodes deployed, each sensing temperature, humidity, barometric pressure, and ambient light. Two nodes are equipped with additional soil moisture probes, and one is measuring wind speed and direction. Each node is powered by 3 AA batteries, with

[15]http://tidmarshfarms.com/

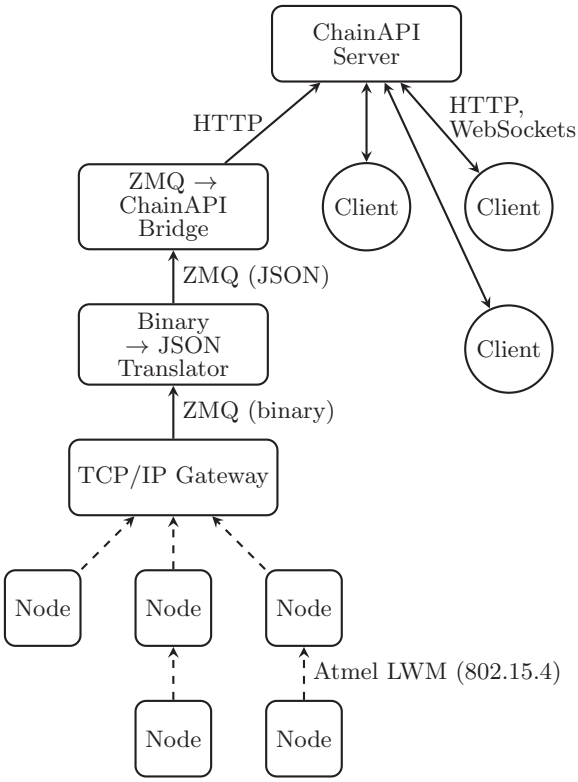

Figure 5: Tidmarsh Architecture

an expected battery life of 2 years, sampling every 30 seconds. There are also several solar-powered nodes acting as repeaters for the mesh network. Figure 5 shows an overview of the Tidmarsh architecture.

11.1 Sensor Infrastructure

One of the goals of ChainAPI is to accommodate a wide variety of heterogeneous underlying sensor systems and provide easy integration. In the case of Tidmarsh there was an existing sensor installation that exposed the sensor data via ZMQ. The sensors communicate with a base station using Atmel's Lightweight Mesh protocol over 802.15.4. Nodes not within direct communication range of the base station can route messages through solar-powered router nodes, which are also collecting data. The base station serves as the TCP/IP Gateway, which receives the RF messages and sends them over WiFi to server via ZMQ. The message payloads are left in their binary format and they are received by the Binary to JSON translator, which parses the binary messages and generates JSON representations of the data. The Binary to JSON translator acts as a ZMQ server, and the JSON messages are sent to any connected clients.

11.2 ChainAPI Representation

To integrate the Tidmarsh sensor installation into ChainAPI, we wrote a small bridge service that connects via ZMQ to the Tidmarsh server and receives JSON messages on each sensor measurement. The only URI needed by the service is the address of the Tidmarsh Site resource on the ChainAPI server, which is passed as an argument on initialization. On initialization the bridge service first requests a list of all devices on the server. Recall from section 10.1 that a device is a collection of sensors in the same enclosure, so in this case each Tidmarsh Node is a device. Within the Tidmarsh system each node has a unique 2-byte identifier, which we use as the device's name field within ChainAPI. From the device and sensor information that the bridge receives from the server, it builds a hash that it can use to look up the URI for a given sensor data as it arrives from the Tidmarsh server. It then subscribes to the ZMQ feed from the Tidmarsh server. As new data come in, the bridge simply looks up the appropriate URI to post the new data to. If data arrives from a sensor or device that the bridge does not recognize, the bridge creates the necessary resource before beginning to post data. This has the benefit that as new sensors come online at Tidmarsh, their data immediately begins flowing into ChainAPI.

Figure 6: Tidmarsh Unity3D Client

11.3 Client Behavior

We developed an interactive 3D client built on the Unity3D game engine[16] to explore and experience the data from the sensor network. As with the Tidmarsh bridge, the client begins by requesting a summary of the site, which includes all devices and sensors at that site. The summary also includes geographic location for each device, which the client uses to place each sensor in the 3D environment. Similar to the bridge, this client builds a hash to map incoming data, but now in the opposite direction (mapping URIs to in-game objects). New data that comes in from the WebSocket subscription will have a link to the containing sensor, so the client needs to quickly map that URI into an in-memory object representing the sensor. As the client parses the summary of devices and sensors at the site, it builds up such a map and instantiates the objects in the game world. As new data comes in, the client looks up the game object in the hash and updates the sensor values. In this client the incoming data is visualized and displayed spatially on a realistic representation of the real-world topology (see Figure 6). The client also incorporates generative music that is driven by the incoming sensor data.

To maintain portability and security, working within the Unity3D environment places substantial constraints on library availability. Here the choice to use standard web technologies sped up development time considerably. With no

[16] http://unity3d.com/

tooling or language support beyond standard HTTP, WebSocket, and JSON parsing we were able to quickly interface with the ChainAPI server and access the data in realtime.

12. FUTURE WORK

A general pattern has emerged in the clients we've built where they often begin with one or more requests for the current state of the resources of interest, after which they subscribe to push updates. The gap between the initial state query and the push updates presents a potential for lost data. One solution would be for the client to subscribe to the updates before requesting the initial state, thus pushing responsibility for merging out-of-order data to the client. Because this pattern is so common a better solution is desirable. Because the initial state information and the WebSocket link for updates are often in the same response, we should be able to include timestamp information in the WebSocket link itself, which the WebSocket server can use on a new client connection to send any messages that otherwise would have been lost. Because the clients treat link URIs as opaque we can add this functionality to the server at a later time without changes to the client.

The ontology that has arisen from our applications is underdeveloped and ad-hoc. While we have discussed how a shared vocabulary could be implemented within our system, more research is necessary to determine a widely-usable set of relations, preferably based on existing ontologies and standards.

Much of this work is predicated on the assumption that we should strive to simplify the client/server interfaces in Web of Things to drive adoption. To validate this assumption we need to build on our qualitative experience with formal user studies.

We have not yet explored the best security models to apply to this architecture. Building on existing web technologies gives us access to a wealth of proven security tools and paradigms, but we need to work on implementing more fine-grained access control while maintaining the scalability advantages such as caching. In addition to access control, there are myriad privacy concerns, as well as questions of tracking data provenance through these layered, multi-tiered systems [4]. There have been great strides in these areas [10] but there are still many open research questions, as well as opportunities to integrate with existing systems.

Efficiently querying sensor data at large time scales will require working with the data at multiple resolutions. Downsampling the raw data brings up many questions that require further research including effective ways to handle missing data. Sampling and interpolation questions extend into space as well, and our work provides a framework for virtual sensors [27] to be created and linked together with the raw data.

As discussed in Section 8, the Web provides a successful model to handle massive decentralized growth in the form of search engines. A collection of hyperlinked sensor data forms a substrate for a more sophisticated ecosystem of crawlers, aggregators, portals, etc. that can be extremely loosely-coupled to the underlying data. It also enables these engines to index the data to optimize for different use cases. Exploring this space is a rich avenue for future research.

13. CONCLUSION

In this work we have introduced a number of design principles that can be used together to build Web of Things applications that are extensible and scalable while maintaining low barriers to entry and interfaces that are familiar to the modern web developer. We have described ChainAPI, our implementation demonstrating and validating that these ideas can support quick development cycles for sensor network applications, and easily integrate with existing infrastructure. We have achieved these goals using almost entirely existing protocols and standards brought together in a unified architecture based on REST and Hypermedia principles. This approach also supports data sharing and interoperability through a shared vocabulary of link relations. With such a proliferation of standards and protocols for the Internet of Things the challenge becomes integrating the pieces into a coherent whole, and ChainAPI is a concrete step in that direction.

14. ACKNOWLEDGMENTS

We would like to thank Cisco Systems, who provides partial funding for this project.

15. REFERENCES

[1] N. K. AIT, J. E. B. AL, and A. G. AL. D2. 3 openiot detailed architecture and proof-of-concept specifications. 2013.

[2] M. Aldrich, A. Badshah, B. Mayton, N. Zhao, and J. A. Paradiso. Random walk and lighting control. In *Sensors, 2013 IEEE*, pages 1–4. IEEE, 2013.

[3] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[4] M. Balazinska, A. Deshpande, M. J. Franklin, P. B. Gibbons, J. Gray, M. Hansen, M. Liebhold, S. Nath, A. Szalay, and V. Tao. Data management in the worldwide sensor web. *IEEE Pervasive Computing*, 6(2):30–40, 2007.

[5] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[6] M. Birbeck and S. McCarron. CURIE syntax 1.0: A syntax for expressing compact URIs. Working group note, W3C, December 2010. Accessed August 8, 2014.

[7] A. Bröring, P. Maué, K. Janowicz, D. Nüst, and C. Malewski. Semantically-enabled sensor plug & play for the sensor web. *Sensors*, 11(8):7568–7605, 2011.

[8] M. Collina, G. E. Corazza, and A. Vanelli-Coralli. Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In *Personal Indoor and Mobile Radio Communications (PIMRC), 2012 IEEE 23rd International Symposium on*, pages 36–41. IEEE, 2012.

[9] M. Compton, P. Barnaghi, L. Bermudez, R. GarcíA-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, et al. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.

[10] Y.-A. de Montjoye, E. Shmueli, S. S. Wang, and A. S. Pentland. openpds: Protecting the privacy of metadata through safeanswers. *PloS one*, 9(7):e98790, 2014.

[11] C. Dixon, R. Mahajan, S. Agarwal, A. B. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*, volume 12, pages 337–352, 2012.

[12] G. Dublon and J. A. Paradiso. Extra sensory perception: How a world filled with sensors will change the way we see, hear, think and live. *Scientific american*, 311(1):36–41, 2014.

[13] G. Dublon, L. S. Pardue, B. Mayton, N. Swartz, N. Joliat, P. Hurst, and J. A. Paradiso. Doppellab: Tools for exploring and harnessing multimodal sensor network data. *Sensors*, pages 1612–1615, 2011.

[14] M. Eid, R. Liscano, and A. El Saddik. A universal ontology for sensor networks data. In *Computational Intelligence for Measurement Systems and Applications, 2007. CIMSA 2007. IEEE International Conference on*, pages 59–62. IEEE, 2007.

[15] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[16] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and content. RFC 7231, IETF, June 2014. Accessed July 24, 2014.

[17] J. J. Gibson. The theory of affordances. *Hilldale, USA*, 1977.

[18] N. Gillian, S. Pfenninger, S. Russell, and J. A. Paradiso. Gestures Everywhere: A multimodal sensor fusion and analysis framework for pervasive displays. In *Proceedings of The International Symposium on Pervasive Displays*, page 98. ACM, 2014.

[19] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.

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

[21] M. Kelly. JSON hypertext application language. Internet Draft draft-kelly-json-hal-06, IETF, October 2013. Accessed July 16, 2014.

[22] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable cloud services for the internet of things with coap. In *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, 2014.

[23] M. Lanthaler and C. Gutl. A semantic description language for RESTful data services to combat semaphobia. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, pages 47–53. IEEE, 2011.

[24] D. Locke. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks Technical Library, available at http://www.ibm.com/developerworks/webservices/library/ws-mqtt/index.html*, 2010.

[25] D. Pfisterer, K. Romer, D. Bimschas, O. Kleine, R. Mietz, C. Truong, H. Hasemann, A. Kroller, M. Pagel, M. Hauswirth, et al. SPITFIRE: toward a semantic web of things. *Communications Magazine, IEEE*, 49(11):40–48, 2011.

[26] D. J. Russomanno, C. R. Kothari, and O. A. Thomas. Building a sensor ontology: A practical approach leveraging ISO and OGC models. In *IC-AI*, pages 637–643, 2005.

[27] C. Sarkar, V. S. Rao, and R. V. Prasad. No-sense: Sense with dormant sensors. In *Communications (NCC), 2014 Twentieth National Conference on*, pages 1–6. IEEE, 2014.

[28] C. Schwartz. Web search engines. *Journal of the American Society for Information Science*, 49(11):973–982, 1998.

[29] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Rfc 7252: The Constrained Application Protocol (CoAP). *Internet Engineering Task Force*, 2014.

[30] M. Sporny. Google adds json-ld support to search and google now, May 2013.

[31] M. Sporny, G. Kellogg, and M. Lanthaler. Json-ld 1.0-a json-based serialization for linked data. *W3C Working Draft*, 2013.

[32] D. Zeng, S. Guo, and Z. Cheng. The Web of Things: A survey. *Journal of Communications*, 6(6):424–438, 2011.

[33] K. Zyp, F. Galiegue, and G. Court. JSON schema: core definitions and terminology. Internet Draft draft-zyp-json-schema-04, IETF, January 2013. Accessed August 5, 2014.