sMAP v2.0 Proposal and Specification

Stephen Dawson-Haggerty

September 11, 2011

Since the initial development in 2009, sMAP has been used to present data from various instruments inside and outside of the LoCal project at UC Berkeley. Its strengths are that it is easy to consume, easy to implement for new device types, and simple to understand. However, use in the real world has also brought out several places where the design was not fully fleshed out. This second version attempts to fix some of these problems, and additional codify some of the places where we operate only on convention.

Contents

1 sMAP Overview

This section presents an overview of the features of the sMAP profile, as currently specified and implemented as a review and to ground the subsequent discussion in the current state.

At its core, sMAP specifies an organization of HTTP resources, and the contents of these resource as JSON schema. The four top-level resources in the sMAP profile are:

`/data` contains resources for reading and controlling meters, sensors, and actuators.

`/reporting` allows control of periodic reports for syndication.

`/status` contains a single universal universal field specifying if the device data is valid, as well as instrument-specific codes.

`/context` contains any information about the device's relationship to other devices. This includes the device's Global Unique Identifier.

The most-commonly used resources are `data` and `reporting`. Data is organized as three-level deep set of resources, of the form `<point>/<type>/<channel>`. The **point** corresponds to a physical point of instrumentation, such as a particular sensor or weather station. It is common for a single transducer to produce multiple time-series; each of these are mapped to channels. The `type` divides

channels into `meterss`, `sensors`, and `actuators`. Finally a **channel** is used to represent a particular stream of readings; it is a resource containing `reading`, `profile`, `formatting`, and `parameter` JSON objects.

The `reporting` resource allows data consumers to install a listener for changes on part of the resource tree. When new readings are received, a sMAP implementation delivers it to subscribers via an HTTP POST. Subscribers specify the set of resources they are interested in via a URL; sMAP sources may deliver either the entire resource or a subset when new readings arrive.

The final resources are `status` and `context`; neither of these are currently used in any systematic way.

sMAP is sometimes run over SSL; when that is the case, we use client certificates for authentication. The authorization database is currently static (per-source).

2 Terminology

Channel A single timeseries of scalar values produced by a sense point.

Sense point A point of physical instrumentation; a single sensor.

uuid Universal Unique Identifier. A practically unique 128-bit identifer generated according to RFC4122.

3 Use Cases and Paradigms

In this section we present a study of how sMAP has been applied in practice. From these we develop a set of protocol design considerations and identify places where sMAP falls short.

3.1 Instrument Modeling

Originally, the intention was that *a sMAP source represents an instrument*; that is, each instrument would be present on the network as a single sMAP source. This intuition was correct in many cases; for instance, sMAP sources make data from Dent, PQube, and Veris electric meters, Vaisala weather stations, HeatX steam gauges, Omega iSeries condensate meters, and many others using this model of "one sMAP source per instrument."

There are however nearly equally numerous cases where sMAP was used to represent a collection of either homogeneous or inhomogeneous instruments. In the first case we have example like those of the ACme and Hydrowatch nodes. These are both embedded devices running the blip 6lowpan/IPv6 stack and periodically report readings back to a network entity. In these cases, sMAP is deployed as a single application-layer gateway which reports readings from a collection of embedded devices. One reason for this appears to be that since sMAP does not provide a discovery mechanism, it is easier for new streams or

channels to appear on an existing well-known sMAP instance, than to inform all consumers that a new sMAP source representing a single ACme.

In the second case of a collection of inhomogeneous instruments, we have have the example of providing a sMAP interface to a much more complicated system like a BMS. In this case, the system is accessed through another application-level gateway rather than at a lower level, as is typical in other applications. In this case, there is typically little organization of the sMAP feeds along any axis. Furthermore, it is difficult to derive meaningful groupings from the structure of the sMAP hierarchy.

3.1.1 Takeaways

1. For homogeneous instruments, sMAP nearly works; however more support is needed for identifying individual instruments; the UUID should be a property of the sense point rather than the sMAP instance.

2. For inhomogeneous instruments, and particularly for where a sMAP gateway is placed in front of a more complicated legacy system, it would be useful to be able to encapsulate any legacy metadata; both per-sense point and per-channel.

3.2 Archival

Typically the first application of a sMAP source is to feed the data into an archival database for future queries, such as `readingdb6`, OpenTSDB, or StreamFS. These systems have used the subscription functionality available through `/reporting`, while ignoring the data FIFO which is present per-channel (the `profile` resource). All of these adaptors expect to receive all data as an HTTP POST to a well-known URL, which typically includes a component indicating where the data contained in the POST.

The subscription mechanism has been extended to be flexible for partial updates of the subscribed "topic", although HTTP processing still incurs high overhead compared to simpler protocols for sources sampled at a high rate. Archivers typically subscribe to a resource such as `~/data/*/*/*/reading`, which causes them to receive incremental updates as new data is available. Since data the data transmitted as a result of this subscription is a nested JSON object, the consumer must use the string components of resource names to identify which timeseries the data belong to.

Another issue of concern is the fact that current implementation practice causes archivers to loose data whenever the archive goes down or is restarted; i.e. for maintenance. No attempt is made to use the `profile` resource to fill in gaps cause by downtime.

3.2.1 Takeaways

1. Since streams are canonically identified by their location in the resource tree, it is not possible to rename them. Thus, in the future, streams should

either be given identities independent of their location (via a UUID or some other mechanism) or their location in the tree should be defined to be static and unchanging.

2. We should examine if better support can be provided for allowing consumers to backfill their databases based on data buffered at the source. For instance, it could be a recommendation to always read part of the `profile` resource at startup to obtain as much missing data as possible.

3. Finally, support could be added for failing over between multiple data collectors; this could potentially allow data to keep flowing in the face of certain types of failures of the archivers. It is unclear weather this is necessary as backups could simply install multiple reporting instances.

3.3 Residential Deployments

As part of the MELS collaboration with LBNL, we have conducted a number of residential pilot studies using the ACme plug-load metering system. The primary unresolved challenge with using sMAP in this environment is the fact that home Internet connections can be very unreliable. In one deployment, external connectivity was available for only a few hours a day, on average.

In another case, the Center for the Built Environment (CBE) deployed a number of sensors on a cart, and included a sMAP frontend for accessing the data. The cart is connected to the Internet via a cellular modem, which its associated benefits and problems; the unifying thread here is that a limited form of intermittent connectivity is a common use case.

In the case of ACmes, the devices themselves perform no buffering and so the sMAP daemon would be responsible for providing the persistent buffer. In the CBE case, the daemon is in truth sitting in front of a MySQL database, and so no data storage is necessary.

3.3.1 Takeaways

1. sMAP should support intermittent connectivity, in the limited sense that a sMAP instrument may not be able to connect to the Internet for a period of time. This could be achieved by adding a "bulk reporting" interface, allowing a sMAP source to buffer writes and push them out when connectivity is available.

2. This bulk load interface should be flexibly implemented so it can either provide data storage or interface with an external storage manager like MySQL.

3.4 Database-backed Deployments

Another common use case has been integration or importing pre-existing databases. For instance, the `obvius.com` site contains whole-building energy data from

much of the UC campus from the past several years. It is very typical for implementors to first acquire real-time access to the data, and then obtain access to a database which contains historal readings. The current sMAP protocol is an inefficient way to upload data since it essentially requires one HTTP `POST` per data point.

3.4.1 Takeaways

1. It should be efficient to bulk-load data via a sMAP-like interface into a database without requiring the sMAP source to be "online."

3.5 Reusable Experiments

Another outcome of the CBE implementation is that they have a small set of instruments which conduct a large number of short experiments. This contrasts with the more open-ended nature of many of our other deployments, where instruments are placed *in situ* and then remain there often for years at a time collecting data as part of the same logical deployment.

The requirement in the CBE case is for there to be a binding between a subset of a timeseries and a logical concept of an "experiment."

1. This functionality seems better implemented on top of the durable identifiers sMAP should provide.

3.6 Actuation

Although activation was originally envisioned as part of sMAP ("that's what the 'A's for!"), it was only recently when any number of actuators became present. The actuation currently present are (a) LabJack devices, (b) ACme X1's with a solid-state relay, and (c) Raritan programmable thermostats. To support these we have modeled several types of actuators: binary, N-state, and continuous. Binary actuators have two positions corresponding to logically "on" and "off"; N state actuators have a number of discrete positions, while continuous actuators can be set to any position within a fixed interval.

Aside from the minor detail that the state can be written in addition to read, actuators have much in common with other channels; units, a current value. It is convenient for each control input to be logged with the same subscription framework as is present for other channels.

3.6.1 Takeaways

1. The activation specification should be documented and cleaned up.

2. The use of SSL to provide authentication, and external databases to provide authorization should be documented.

3.7 Discovery, Organization, and Management

The data sources we deal with are typically very disorganized, and so we have not encountered problems where the originating data source's information model was so sophisticated that we could not map it onto a sMAP resource hierarchy. However, the current "state of the art" for locating new sMAP sources is to manually enter their IP address into a relational database. The HTTP hierarchy and SSL certificates for a particular sMAP source are manually configured (by the sMAP source author).

The primary question we face here is whether the current approach is sufficient, or if we should specify a more complicated protocol with capabilities for discovery and presence detection like XMPP, Zeroconf, or SNMP. Both of these protocols come with both costs and benefits; for instance with a Zeroconf-based approach, we could automatically discover the service locations of sMAP

3.7.1 Takeaways

1. Include a specification for performing service discovery over Zeroconf. This means defining canonical MDNS names such as
   `Smap Sever 2._smap._http._tcp.berkeley.edu`.

2. Specify how sMAP services may be combined via a reverse-proxy mechanism in order to aggragate a large number of sMAP sources into one logical source.

4 Protocol Modifications

Based on the use cases and recommendations in section **??**, we make the following high-level changes to sMAP.

4.1 Data Representation

1. The primative streams we are representing are Timeseries. Time series consist of sequences of readings from a single channel of an instrument and associated metadata. Time series may be organized by the sMAP implementer into Collections, representing logical organizations of the instrumentation; this organization is reflected in the resource heirarchy exposed by the sMAP HTTP server.

2. sMAP 2.0 supports adding Metadata to either Timeseries or Collections to better support integrating existing data sources where the Metadata should be transfered along with the data.

3. The only objects sMAP represents are Timeseries and Collections. Timeseries are durably identified by `uuid`s.

4. Schemas are expressed as Avro schema. If clients indicate that they can accept this schema via the Accepts header, servers may return Avro-packed results with the appropriate Content-encoding set. Clients should be made available to take advantage of this. This permits the efficient transfer of large numbers of objects.

4.2 Metadata

sMAP v1 supported very limited metadata. In version 2.0, we extend the metadata model in oder to better support integrating existing systems, as well as building systems where the implementors have significant metadata they wish to tag their data with. We make several simplifying assumptions:

1. Metadata only has meaning when it is attached to data, where "data" is one or more time-value pairs (data points). We logically consider each data point to be attached to a full set of metadata.

2. Metadata is inherited within a sMAP source: the metadata for any data point consists of the Metadata for that Timeseries, along with the Metadata all Collections in the recursive parent set. If the same piece of metadata is present at multiple places, the metadata "closest" to the Timeseries (*i.e.*, deepest in the tree) is used.

With this system, we can both efficiently compress the metadata when transmitted over the network and also resolve many issues dealing with what happens when resources move: we do not need to track the move since instead all timeseries which were moved simply receive new metadata. Since the Metadata can only apply to actual data, there is never an ambiguity about which piece of metadata applies to a point, and it is always safe to re-send all metadata.

4.3 Reporting

1. Modify the `reporting` hierarchy to be a collection, and eliminate the `reports` and `create` sub-resources, which are not RESTful.

2. Reports should allow the provisioning of a list of delivery locations, which will be attempted in order.

3. Provision should be made for "reliable reporting," when feasible. sMAP servers should treat the stream of outgoing readings as a log, and only truncate the log when it has been successfully delivered to the endpoint.

4.4 Aggregation and Proxies

Add the ability for a sMAP proxy to suck up a bunch of other sMAP sources discovered by Zeroconf and republish them as a single sMAP feed as a reverse proxy. This "aggregates" a large number of small sources into one larger one and paves the way for how we would take a large number of, say, ACme feeds

into a manageable, discoverable number of feeds at the aggregate. The proxy should buffer received readings so that it can run its own reporting engine.

5 Detailed Specification

The root-level resources sMAP provides are modified to include only:

`/data` Contain all Timeseries and Collections served by this sMAP server.

`/reports` Resource responsible for exposing reporting instances.

`/actuate` If present, the sMAP source supports actuating multiple points at the same time.

5.1 Data Representation

Each resource returned by a sMAP server is encoded as a JSON object; optionally encoded using Avro.

5.2 Timeseries Representation

Each instrument channel is expressed as a single `Timeseries` object. Each time series is placed into the sMAP resource heirachy in a location determined by the sMAP implementor.

```
{
  "name" : "Timeseries",
  "type" : "record",
  "namespace" : "edu.berkeley.cs.local",
  "doc" : "A container class containing a single time series",
  "fields" : [
              {"name" : "uuid", "type" : "uuid"},
              {"name" : "Description", "type" : ["null", "string"]},
              {"name" : "Metadata", "type" : ["null", "Metadata"]},
              {"name" : "Properties", "type" : ["null", "Properties"]},
              {"name" : "Actuator", "type" : ["null", "Actuator"]},
              {"name" : "Readings", "type" : ["null", {"type" : "array", "items" : "Reading
          ]
}
```

Description a string description of the channel.

Properties information about the channel required to store or display the data. This is made up of the data type, engineering units, and time zone.

Metadata additional information about the channel which is not necessary to archive or properly display readings. This is provided to facilitate the integration of existing sources. The full set of metadata for a timeseries also includes all the metadata for Collections in its recursive parent set.

Actuator if present, the channel includes an actuation component. The object describes what actuation is possible.

Readings a vector made up of the latest readings from the instrument. The data type of the `ReadingValue`s must match the type specified in the `Parameter` object.

```
{
 "name" : "ReadingValue",
 "type" : "record",
 "namespace" : "edu.berkeley.cs.local",
 "doc" : "Represent a reading from a single stream",
 "fields" : [
   {"name" : "ReadingTime", "type" : "long"},
   {"name" : "ReadingSequence", "type" : ["null", "long"]},
   {"name" : "Reading", "type" : ["long", "double", "string"]}
   ]
}
```

Individual reading points are made up of `ReadingValue` objects. Each point must include a timestamp, the reading, and an optional sequence number. The timestamp must be in units of Unix milliseconds. [1] Implementations should advance time in increments meaningful to the underlying process and taking into account the resolution of the clock source.

5.3 Collection Representation

sMAP 2.0 expects implementors to organize the Timeseries into collection which reflect properties of the underlying instrumentation. For instance, three-phase electric meters will mostly likely group timeseries by phase, while a system collecting data from a large number of ACme plug-load meters would group timeseries by individual ACme and also gateway location.

```
{
  "name" : "Collection",
  "type" : "record",
  "namespace" : "edu.berkeley.cs.local",
  "doc" : "A container class containing other collections or timeseries",
  "fields" : [
           {"name" : "Proxy", "type" : ["null", "boolean"], "default": "false"},
           {"name" : "Metadata", "type" : ["null", "Metadata"]},
```

[1] Unix timestamps aren't actually unambiguous, due to complications arising from leap seconds. It seems like the alternative to using this time representation would be to use the ISO 8601:2004 time format as suggested by RFC3339. However, these string values are rather large when transfering a large number of readings so it's unclear whether the resulting object would be compact enough to satisfy our needs. If it is REQUIRED, it additionally imposes higher burden on embedded devices who must maintain a calendar instead of simply a RTC since 1970.

```
            {"name" : "Contents", "type" : { "type" : "array", "items" : "string"}}
  ]
}
```

The `Collection` object holds information about these collections. A `Metadata` object may be included to capture metadata which applies to all timeseries subordinate to the collection, rather than just one of them. The only other object in the collection is the `Contents` list, contains string names of the collection elements.

Clients may fetch the collection elements by adding a string name from the `Contents` list to the resource name where the collection is located. The names in the map may not use relative URL operators such as `.` and `..`, and should be quoted before attempting to fetch the resource.

5.4 sMAP URI

URIs consist of six components:

`<scheme>://<netloc>/<path>;<params>?<query>#<fragment>`

sMAP does not use the `params` or `query` components; the other components are used as follows:

scheme Either `http` or `https`, when run over SSL.

netloc The network location of the sMAP server.

path The resource path, separated by '/' characters. Top-level resources are `data`, `reports`, and `status`.

fragment Used as a sub-resource selector to select only part of a resource. For instance, given the fragment `#Reading`, the sMAP server should return only the `Reading` key of the JSON resource at the given `path`. In addition, the `uuid` field of the containing object is also included.

5.4.1 Fragment Selection

Fragment selection can be used to return only part of an at a particular resource location. The fragment component of the URI begins with a hash (`#`), and consists of a single string keyname. When in use, the returned object includes the all "required" keys as determined by the Avro schema, and the keys named by the fragment selector. Support for this feature is optional.

5.4.2 Recursive Descent

Individual resources on sMAP servers may be queried by asking for them by their path name; sMAP implementers are responsible for determing the appropriate organization of the resources under `data`. If a single object is identified, that is returned to the client.

sMAP servers should support a resource to allow clients to efficiently collect an entire portion of the resource heirarchy. The resource in question is `+`. When retrieved, it should return a map of all paths on the server recursively found under that resource.

To give an example, consider a sMAP server with one sensor, located at `/data/sensor0/channel0`.

The result of performing a GET on `/data/+` might be:

```
{
  "/" : { "Contents" : ["sensor0"] },
  "/sensor0" : { "Contents" : ["channel1"] },
  "/sensor0/channel0" : {
     "uuid" : "2aa7a830-a348-11e0-87e9-0026bb56ec92",
     "Readings" : [ {"ReadingTime" : 0, "Reading" : 12.5 } ]
  }
}
```

Since the entire heirarchy is contained in this resource along with all metadata, the client can unambiguously determine which metadata applies to each timeseries; in this case no metadata was included for brevity, but it could have present in any of the three objects.

If this resource is requested at a subordinate level (for instance `/data/sensor0/+`, the return object's root path should be rooted at the relative url (`/data/sensor0`) and implementors should be aware that they will not know about any metadata higher in the hierarchy.

5.5 Reporting

The purpose of the reporting is to allow consumers to receive timely notifications of changes to the sMAP tree. In principle, it should allow consumers to receive a copy of every version of an object; however, sMAP servers may choose which changes to publish.

Clients wishing to receive callbacks about changes do so by creating a reporting instance. They do so by posting a reporting object to the `/reports` resource on the server; the schema of the body is

```
{
  "name" : "Reporting",
  "type" : "record",
  "namespace" : "edu.berkeley.cs.local",
  "doc" : "A single report instance",
  "fields" : [
              {"name" : "uuid", "type" : "uuid"},
              {"name" : "ReportResource", "type" : "string"},
              {"name" : "ReportDeliveryLocation", "type" : {"type" : "array", "items" : "str
              {"name" : "MinPeriod", "type" : ["null", "long"]},
              {"name" : "MaxPeriod", "type" : ["null", "long"]},
```

```
            {"name" : "ExpireTime", "type" : ["null", "long"]}
        ]
}
```

All time fields are specified in millisecond units.

`ReportResource` identifies the resource on the server which the client is interested in receiving, relative to the `/data` resource.

`ReportDeliveryLocation` is a list of URIs specifying where report data should be delivered to. The sMAP server SHOULD continue attempting to deliver data until it recieves an HTTP success response from one of these servers. It should also attempt delivery to servers in the order they are present in this object.

`MinPeriod` specifies the minimum interval between reports. A sMAP server should not deliver reports more frequently than this. If the minimum period is so long as to prevent the sMAP server from buffering all the data accumulated in the period, it should deliver the latest data. If not included, the default is 0.

`MaxPeriod` specifies the maximum period between reports. After this much time elapses, the sMAP server should deliver a report regardless of whether there is new data to indicate liveness. If not included, the default is ∞.

`ExpireTime` time in UTC milliseconds after which reports should be stopped, undelivered data dropped, and the report removed. Default is "never."

A copy of one of these objects installed in a server is known as a **reporting instance**. The most common use case is that a sMAP client subscribes to a resource like `/+` to recieve all new data from the source. sMAP implementations MAY choose not to deliver the entire `Timeseries` object but instead only include keys which have changed – typically only `Readings`. Furthermore, given a series of `ReadingValues`, the server may combine them into any number of deliveries.

Creating reports New reports are created by POSTing the above object to the `/reports` resource. The sMAP server MAY restrict the listing of reports (`GET /reports`) to authenticated clients, or not support it at all. If the reporting instance is created successfully, it should be located at `/reports/<uuid>`, and can be removed using the HTTP `DELETE` verb. The server may support modifying it by PUTing a new copy of the report instance to `/reports/<uuid>`.

Authentication If authentication and authorization were in use when the report was created, the sMAP source MAY require the same authentication and authorization to modify or delete the report.

Failure sMAP servers may implement a number of policies to deal with reporting instances where the delivery location is not accessable. They may wish to keep trying for a fixed period of time before removing or deactiving the report instance, or to buffer data while the recipient is down and retry with all of the accumulated data periodically.

5.5.1 Static Report Configuration

It is intended that sMAP will mostly be used in an "online" setting, where sources are automatically discovered and subscribed to by data consumers or other sMAP proxies. However, there are certain use cases where it is desirable for reporting to be configured manually on the sMAP server, rather than using the online system. For instance, if the sMAP server is behind a NAT, or if the data is actually not online but rather an import of an existing database. In these cases, a sMAP server may provide a provision for configuring report instances via a config file or some other mechanism, rather than using the online mechanism.

5.5.2 Differential Transmission

When a report instance is created, a sMAP source MUST send the entire resource which the report instance refers to. However, frequently only a small portion of the resource changes; for instance, the metadata is often static and only the Readings part of a timeseries changes with each reading. For a particular report instance, a source may only send changes once it has successfully delivered the entire object.

5.6 Metadata

Both `Timeseries` and `Collections` support the inclusion of `Metadata` to provide additional information about measurements. Ideally this metadata is pragmatically obtained from an existing system, or entered by the implementator; it is not intended that this metadata will change frequently. For convenience, sMAP 2.0 defines two types of metadata to facilitate information exchange: `Instrument` and `Location` metadata. All other information must be placed into the `Extra` fields, and may encapsulate any other metadata description in use.

```
{
  "name" : "Metadata",
  "type" : "record",
  "namespace" : "edu.berkeley.cs.local",
  "doc" : "A container class containing a single time series",
  "fields" : [
    {"name" : "SourceName", "type" : ["null", "string"]},
    {"name" : "Instrument", "type" : ["null", "InstrumentMetadata"]},
    {"name" : "Location", "type" : ["null", "LocationMetadata"]},
    {"name" : "Extra", "type" : ["null", { "type" : "map", "values" : "string"} ] }
  ]
}
```

The most important simplification to metadata handling in sMAP is the statement that *metadata applies to data points*; that is, a metadata change to

a collection which has no contents has no effect. The goal of a sMAP server is to allow consumers to compute the full set of metadata which applies to any data point. It is always safe to send all metadata, which is included when a client fetches `/data/+`. To allow compression, a consumer of a sMAP source may never assume that metadata applies to any resource out of the document where it was received.

5.7 Actuation

If actuation is present, the `Timeseries` resource corresponding to the actuator must include a `Actuator` key. The actuator key models the type of actuation supported by the control point. Three control models are supported:

binary actuators have only two positions, corresponding to logical "on" and "off". Must use the `long` data type.

discrete actuators can take a finite number of values. May be either `long` or `double` valued.

continuous actuators can take any value in a range. Must be `double` valued.

```
{
  "name" : "Actuator",
  "type" : "record",
  "namespace" : "edu.berkeley.cs.local",
  "doc" : "A representation of the type of actuation possible",
  "fields" : [
    {"name" : "ActuatorModel", "type" : {"type" : "enum", "symbols" : ["binary", "discrete",
                                          "name" : "ActuatorModels"}},
    {"name" : "MinValue", "type" : ["null", "long", "double"]},
    {"name" : "MaxValue", "type" : ["null", "long", "double"]},
    {"name" : "StepSize", "type" : ["null", "long", "double"], "default": 1},
    {"name" : "Values", "type" : ["null", {"type" : "array", "items" : ["null", "long", "dou
  ]
}
```

For binary actuators, the `Actuator` object should not include any other keys. Discrete actuators define a set of possible values using the union of `MinValue`, `MaxValue`, `StepSize`, and the `Values` array.

$$[\texttt{MinValue} : \texttt{MaxValue} : \texttt{StepSize}] \cup \texttt{Values}$$

For continuous actuators, the proceedure is the same with the exception that `StepSize` is not used and the interval is treated as continuous.

When a `Timeseries` is used to represent an actuator, the `Readings` field should be used to communicate the current actuator position. The sMAP implementor may choose to only generate a new reading when a control input is received, periodically, or when the state of the actuator changes.

5.7.1 Transmission of Actuation State

Clients wishing to affect actuator position do so by `POST`ing a `Command` object to the `Timeseries` representing the actuator. The `uuid` in the `Command` must match the id of the resource.

sMAP servers providing actuation may also provide the `/actuate` resource, which supports only the `POST` verb. The function of this resource is to allow clients to control multiple actuators at the same time, by POSTing a list of `Commands`.

```
{
  "name" : "Command",
  "type" : "record",
  "namespace" : "edu.berkeley.cs.local",
  "doc" : "A command to change the state of the actuator",
  "fields" : [
    {"name" : "uuid", "type" : "uuid"},
    {"name" : "Value", "type" : ["long", "double"]}
  ]
}
```

5.8 Authentication and Authorization

sMAP servers may support a variety of schemes for authentication and authorization. Although a detailed specification of how these are to be employed, we envision that implementors may want to use mechanisms such as HTTP Basic or Digest, HTTPS, or oauth to secure communication with sensors and control access to actuation points.

In following good design discipline, sMAP servers should separate *authentication* from *authorization*. For instance, at Berkeley we authenticate clients using HTTPS with client certificates to authenticate clients, and look up client authorization in a database to provide authorization to perform a certain action. sMAP servers should use HTTP mechanisms to communicate to the client that additional authorization or authentication is required to perform an action.

The situation becomes more complicated when proxies are in use – although HTTP allows the Basic and Digest authentication to be passed through a proxy within an HTTP session, other mechanisms such as HTTPS may not support this.

5.9 Avro Transmission over HTTP

All content transfered between sMAP clients and servers must be encoded as JSON objects; the appropriate `Content-type` is `application/json`.

If the client explicitly asks for Avro by sending an `Accept-Encoding:` HTTP header which includes the `x-avro` encoding type, the sMAP server may reply with a response encoded with Apache Avro. The server may also apply additional compression such as gzip or deflate if support is indicated.

When Apache Avro is returned by a sMAP server or sent as part of a POST by a client, the sender must provide sufficient information to the recipient to interpret the response, using HTTP headers. In particular, the server or client must indicate that Avro is in use by adding a `Content-Encoding: x-avro` header in addtion to other encodings present. The sender must also indicate the Avro schema name the content is encoded with by adding an `X-Avro-Schema` header; the value of the header should be the full name of the schema in use (namespace and name).

Finally, servers supporting Avro must provide the ability for clients to download the set of schemas needed to communicate with the server. They should do this by providing a well-known resource, `/avro-schemas` which contains a JSON list of all schemas in use by the server. Servers may provide additional schemas at any level of the resource hierarchy, with these schemas being used to interpret responses at any level below the level at with that resource is found. All content transfered between client and server must reference a schema found in one of these documents; in particular, if the body of a client's HTTP POST is Avro encoded, the schema must be found in one of these documents.

The `avro-schemas` resource should never be Avro-encoded, but servers SHOULD use HTTP cache-validation mechanisms (`Expires` or `Etags`) so clients can avoid excessive transfers of this resource.

5.10 Proxying of sMAP Sources

5.11 Discovery using Zeroconf

Revision History

6/23/2011 Initial revision