Qualia
networks incorporated

Chariot: web-of-things shield for Arduino

Standard resources made available by Chariot-equipped Arduinos (aka *motes*)

The table below describes core resources created for web-of-things access by the Chariot shield for Arduino. The listed arduino/ hardware resources make Arduino's pins available for reading, writing, mode setting and observing from a distance. Similarly, standard hardware resources of Chariot make it possible to acquire system temperature, movement, shock, direction, RF signal quality, and battery voltage. Software resources allow sketches on the mesh and clients anywhere to list and search resources and control low power sleep behavior.

A set of serial commands allows Chariot to be reconfigured from the Arduino serial monitor. Transmit power, channel selection, PAN ID and address values in permanent storage may be changed. Additionally, health and mote visibility status may be requested. See section on serial interface below for more details.

Note: Chariot's API provides the capability for creating and naming new dynamic and event-driven resources in your sketches. New resources can be constructed from existing ones in order to extend capabilities. For example, digital pins are observable on a timed basis only, by default. For example, a sketch can create a new resource, /event/digital that triggers, alerting observers, when a pin toggles.

Table 1. Chariot core resources

Resource	Description	RESTful functions	Observation types
	Hardware		
/arduino/analog	Reads the value from the specified analog pin. Writes an analog value (PWM wave) to a pin.	GET, POST, PUT, OBS and DEL. POST and PUT provide the same function. DEL deletes an observer (the requestor).	Periodic. Frequency is settable, default is 10 seconds. Use PUT or POST to modify frequency.
/arduino/digital	Reads or writes pins on the configured as either inputs or outputs.	GET, POST, PUT, OBS and DEL. POST and PUT provide the same function. DEL deletes an observer (the requestor).	Periodic. Frequency is settable, default is 10 seconds. Use PUT or

			POST to modify frequency.
/arduino/mode	Set the mode for a digital pin: INPUT, OUTPUT, or INPUT_PULLUP. (See the digital pins page for a more complete description of the functionality).	GET, POST, PUT. POST and PUT provide the same function.	N/A
/sensors/battery	Provides operating Vin from Chariot MCU voltage sensor.	GET	N/A
/sensors/radio	Provides RSSI and LQI link values from Chariot's 802.15.4 radio. These measure receive signal strength and error performance of the shield.	GET, POST, PUT, OBS and DEL. POST and PUT provide the same function. DEL deletes an observer (the requestor).	Periodic. Frequency is settable, default is 10 seconds. Use PUT or POST to modify frequency.
/sensors/tmp275-c	Provides the temperature in C from the TI TMP275-C on board Chariot.	GET, POST, PUT, OBS and DEL. POST and PUT provide the same function. DEL deletes an observer (the requestor).	Periodic. Frequency is settable, default is 10 seconds. Use PUT or POST to modify frequency.
/sensors/fxos8700cq-c/mag	Provides Chariot's magnetometer reading (NXP 8700cq-c).	GET, POST, PUT, OBS and DEL. POST and PUT provide the same function. DEL deletes an observer (the requestor).	Periodic. Frequency is settable, default is 10 seconds. Use PUT or POST to modify frequency.
/sensors/fxos8700cq-c/accel	Provides Chariot's accelerometer reading (NXP 8700cq-c).	GET, POST, PUT, OBS and DEL. POST and PUT provide the same function. DEL deletes an observer (the requestor).	Periodic. Frequency is settable, default is 10 seconds. Use PUT or POST to modify frequency.

	Software		
/search	Search a visible mote's resources for a full or partial match on the name provided.	GET	N/A
/sleep	Put Chariot to sleep for the number of seconds specified, or if no value is specified, into a power down sleep that requires signaling by the Arduino for wakeup.	PUT	N/A
/.well-known/core	This is a standard CoAP feature that returns a list of the target mote's resources and the attributes of each. Included in the list are also any dynamic resources created in the sketch via the Chariot library.	GET	N/A

Accessing your things on meshed Arduinos from your sketch and other clients

Getting and setting Arduino analog and digital pin values from another sketch on the web-of-things or app in the cloud

See the Arduino tutorial and reference pages on the use of digital and analog pins. A number of special considerations apply. For example, analog pins may be used as gpio's just as their digital counter parts. Also, some arduino boards have more gpio's than others, for example the Mega2560 has 54 digital pins and 16 analog pins, while UNO has 14 digital pins and 6 analog pins. Mega boards also have 4 hardware serial ports, while UNO has just one. Here are some examples showing how Chariot makes the digital and analog pins of your Arduino available for observation and control on the web:

-GET current value for analog pin 5 from a meshed Arduino (i.e., the one @ chariot.c350f.local)

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "arduino/analog",
       opts = "pin=5";
```

```
ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or from webapp to Chariot:

```
coap://chariot.c350f.local/arduino/analog?get&pin=5
```

Response:

```
chariot.c350f.local: 2.05 CONTENT Pin A5 set to 358
```

-GET current value for digital pin 12 from a meshed Arduino

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "arduino/digital",
       opts = "pin=12";

ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/arduino/digital?get&pin=12
```

Response:

```
chariot.c350f.local: 2.05 CONTENT Pin D12 set to 0
```

-Make digital pin 12 an output on meshed Arduino using PUT

Sketch API:

```
String mote = "chariot.c350f.local",
       Resource = "arduino/mode",
       opts = "pin=12&val=output"; // String option args together with '&'

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/arduino/mode?put&pin=12&val=output
```

Response:

```
chariot.c350f.local: 2.05 CONTENT Pin D12 configured as OUTPUT
```

-PUT a new value to digital pin 12

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "arduino/digital",
       opts = "pin=12&val=1"; // String option args together with '&'

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/arduino/digital?put&pin=12&val=1
```

Response:

```
chariot.c350f.local: 2.05 CONTENT Pin D12 set to 1
```

-PUT the pin back to its original value

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "arduino/digital",
       opts = "pin=12&val=0";

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/arduino/digital?put&pin=12&val=0
```

Response:

```
chariot.c350f.local: 2.05 CONTENT Pin D12 set to 0
```

Hardware resource sharing between Arduino and Chariot shield

Arduino and Chariot use a serial port for the exchange of information. On UNO, 2 digital pins, 11 and 12, are allocated for this—with the UNO creating a software serial port. On Leonardo (including Yun), they are also digitals 11 and 12. Note that Leonardo is no longer supported by Arduino and has been dropped from Qualia support as well. All boards share 3 gpio's that consume digital's 7-9 and also one external interrupt pin which uses digital pin 3. Other boards, such as Mega 2560 have additional hardware serial ports that remove the requirement for using two of the digital gpio's as a software serial channel. In summary, digital pins 7-9 are used for gpio sharing and control between Chariot and Arduino. For UNO and Yun, pins 11 and 12 are also unavailable as they are used for transmit and receive serial channels between Chariot and Arduino. See the file ChariotEPLib.h for the specific pins shared by your Arduino board model.

Chariot builtin sensors

Chariot motes come standard with high performance temperature sensor and a 6-axis accelerometer that includes a magnetometer. Both reside on the I2C bus shared with Arduino and may be accessed from either side. These sensors have fixed resource names for external reference. These can be extended within the sketch through the addition of dynamic, event-driven resources as needed. All of the sensors will output JSON strings if the URL parameter string ";ct=50" is added to the URL. See the Radio sensor example below.

Onboard temperature sensor

The temp sensor is a Texas Instruments TMP275 and is accurate within ±0.5°C. It is an integrated digital temperature sensor with its own 12-bit analog-to-digital converter with a range from −20°C to 100°C. At a resolution of ±1°C (Maximum) its range covers from −40°C to 125°C. It consumes less than 50µA.

The I2C addressing scheme of the TMP275 allows for up to eight devices on a single I2C. For this reason, ours, being chariot-resident is denoted as:

```
/sensors/tmp275-c
```

Any others that are added maybe named at the discretion of the sketch designer. Note also, since the sensors sit on the I2C, no gpio's are consumed for their support. Reading the value is done from an app or mesh neighbor using the URL scheme shown below in the sketch example:

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "sensors/tmp275-c",
       opts = "";
```

```
ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/sensors/tmp275-c?get
```

Response:

```
chariot.c350f.local: 2.05 CONTENT 29.0(C)
```

Note also that temperatures are reported in Celsius. This is because the device operates in this unit of conversion. Your sketch or web-app must convert to any other units, as requirements call for.

Onboard accelerometer

The accelerometer is a NXP FXOS8700CQ, 6-Axis sensor with integrated linear accelerometer and magnetometer. This device is also connected to the I2C bus and consumes no connector pin space for its functions. It is also a low-power sensor, consuming just 80µA at 25Hz with both accelerometer and magnetometer active. This device appears as separate accelerometer and magnetometer resources to external accessors. Both support GET, OBS (timed), and DEL RESTful operations and can be programmed, as with the temp sensor, to perform periodic reporting of their current values:

Accelerometer: /sensors/fxos8700cq-c/accel:

Sketch API:

```
String mote = "chariot.c350e.local",
       resource = sensors/fxos8700cq-c/accel",
       opts = "";

ChariotEP.coapRequest (COAP_GET, mote, resource, APPLICATION_JSON, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/sensors/fxos8700cq-c/accel?get;ct=50
```

Response:

```
chariot.c350e.local: 2.05 CONTENT {"ID":"FXOS8700CQ Accel","X":597,"Y":24,"Z":-4635}
```

And:

Magnetometer: /sensors/fxos8700cq-c/mag:

Sketch API:

```
String mote = "chariot.c350e.local",
       resource = sensors/fxos8700cq-c/mag",
       opts = "";

ChariotEP.coapRequest (COAP_GET, mote, resource, APPLICATION_JSON, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/sensors/fxos8700cq-c/mag?get;ct=50
```

Response:

```
chariot.c350e.local: 2.05 CONTENT {"ID":"FXOS8700CQ Magnetometer","X":14,"Y":342,"Z":32}
```

Important caveat for the fxos8700cq device

These sensors provide many functions beyond the basic ones we address in the RESTful API. You may bind any of these functions to your own dynamically created resources in your sketch and export them over the web for RESTful access by remote sketches and clients. The example sketch, *webofthings_event_trigger*, gives an example of this using the temperature sensor. Like builtin resources, ones you create are completely controllable from the sketch, which is free to provide URL semantics for using the new functionality via GET**,** OBS**,** DEL**,** POST and PUT.

Chariot radio performance parameters

RF parameters RSSI and LQI are commonly used as measures for the wireless link quality. The RSSI (Radio Signal Strength Indicator) provides a measure of the signal strength at the receiver whereas the LQI (Link Quality Indicator) reflects the bit error rate of the connection. Chariot provides both of these in order to aid in positioning Chariot/Arduino motes and antenna selection for optimal radio performance. A technical discussion of these is outside the scope of this document. For that, the reader is directed to the journal article, *RSSI and LQI vs. Distance Measurement, Wireless Sensor Networks and Electronics—Spring 2011*, Aarhus School of Engineering. It can be found online. Understanding the quality of the 6LoWPAN RF link from these two parameter readings is provided therein commonsense terminology.

These two parameters are available from Chariot as the following example shows:

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = sensors/radio",
       opts = "";
```

```
ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/sensors/radio?get
```

Response:

```
chariot.c350f.local: 2.05 CONTENT Lqi=255 Rssi=-44dBm
```

We can ask for the values to be returned coded as JSON by including its type (i.e., generate "ct=50" in the URI):

```
ChariotEP.coapRequest (COAP_GET, motes, resource, APPLICATION_JSON, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/sensors/radio?get;ct=50
```

Response:

```
chariot.c350f.local: 2.05 CONTENT {"ID":"ATMega256RFR2 Radio","Lqi":255, "Rssi":-45dBm}
```

RSSI relates primarily to power at a distance and LQI to packet error rate performance.

Chariot battery sensor

Chariot has a supply voltage monitor that can be used to report its value remotely. The value reported is the operating voltage floor, meaning that the supply voltage is at least the value being reported. Chariot is a 3V3 design; expected values for supply voltage range from approximately 2.55V to 3.68V:

Sketch API:

```
String mote = "chariot.c350e.local",
       resource = sensors/battery",
       opts = "";

ChariotEP.coapRequest (COAP_GET, mote, resource, APPLICATION_JSON, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/sensors/battery?get;ct=50
```

Response:

```
chariot.c350e.local: 2.05 CONTENT {"ID":"Chariot VIN Reading","Val": 3.525,"Unit":"V"}
```

Observing builtin sensors

Most of the builtin hardware sensors (see Table 1 above), are observable on a periodic basis. Analog and digital pins, temp sensor, accelerometer and magnetometer and radio are observable this way. As has been mentioned previously, these sensors must have a period set for them prior to requesting periodic observations.

Let's set up a periodic observation of Chariot's TI TMP275 I2C temp sensor, which is what the sketch is monitoring from the arduino. The default period for the sensor is 7 seconds. This can be changed by URL, as in this example changing it to 4 seconds.

Sketch API:

```
String mote = "chariot.c351f.local",
       resource = sensors/tmp275-c",
       opts = "period=4&units=secs";

ChariotEP.coapRequest (COAP_POST, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c351f.local/sensors/tmp275-c?post&period=4&units=secs
```

Response:

```
chariot.c351f.local: 2.04 CHANGED Period set to 4 seconds
```

Starting a periodic observation

Once the period has been established the observation can be started. The periodic delivery of the sensor value will continue until you stop the observation. Note that other clients may request observations as well. The Chariot target will direct responses to each and every observer until the last observer leaves the subscriber group. Notice that the token ("TKN") value is used by the client to differentiate between one observation and others currently open in the sketch or client.

Sketch API:

```
String mote = "chariot.c351f.local",
       resource = "sensors/tmp275-c",
       opts = "";

ChariotEP.coapRequest (COAP_OBSERVE, mote, resource, APPLICATION_JASON, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c351f.local/sensors/tmp275-c?obs;ct=50
```

Responses:

```
chariot.c351f.local: 2.05 CONTENT TKN=c735b34a {"ID":"Chariot TMP275","Val": 26.8,"Unit":"C"}
chariot.c351f.local: 2.05 CONTENT TKN=c735b34a {"ID":"Chariot TMP275","Val": 26.8,"Unit":"C"}
chariot.c351f.local: 2.05 CONTENT TKN=c735b34a {"ID":"Chariot TMP275","Val": 26.8,"Unit":"C"}

...

chariot.c351f.local: 2.05 CONTENT TKN=c735b34a {"ID":"Chariot TMP275","Val": 26.8,"Unit":"C"}
```

Stopping a periodic observation

Subscriptions to periodic or event-driven observations are cancelled using the DELETE RESTful primitive. For the periodic temperature observation above:

Sketch API:

```
String mote = "chariot.c351f.local",
       resource = "sensors/tmp275-c",
       opts = "";

ChariotEP.coapRequest (COAP_DELETE, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c351f.local/sensors/tmp275-c?del;
```

Response:

```
chariot.c351f.local: 2.02 DELETED Chariot TMP275 observation terminated
```

Creating, managing and observing event-triggered dynamic resources

Chariot provides the sketch writer with the capability of adding up to 8 additional resources on an Arduino-Chariot mote. These have added functionality over the standard resource behavior of those listed by Table 1 above. First, dynamic resources grant exclusive access of DELETE and POST primitives to the creating sketch. Remote sketches and clients may issue GET, PUT and OBServe primitives against these resources. Additionally they may discover them using the resource search facility described below.

Remote sketches and apps use PUT to parameterize the behavior of these resources. This is described in paragraphs below. The number and type of parameters that can be maintained for a dynamic resource is somewhat arbitrary—that is, it is really subject only to the amount of data memory available to the sketch. This is because the parameters are not kept by Chariot. What is kept in Chariot is the single value that characterizes the current state of the resource (e.g., temperature value). The sketch writer has complete control of the construction of the value attribute of the dynamic resource, and can allocate up to 64B to hold it.

Dynamic sketches behave like one shot timers. That is you use PUT or POST to arm them and wait for the arming condition to trigger an event. The event will be seen by all external sketches or clients that have issued an OBServe on this resource. Afterward, the resource returns to its unarmed state. The sketch or client must rearm to create the next event opportunity. Those sketches and clients that have registered subscriptions will receive subsequent event publications until they delete their subscriptions (see section above on deleting observations of periodic resources) or the mote is reset.

Example sketch: "*Chariot_EP_sketch_webofthings_event_trigger*"

The easiest way to illuminate the main concepts and methods regarding dynamic resources is by way of example. The URLs in these examples were issued from a Chrome browser running our html and JavaScript websocket frontend and also from various sketches via use of our Chariot library. Note again that our convention has it that the Arduino owns the POST and DELETE RESTful primitives. This is because the web-of-things event system is designed expressly to enable your sketch to create and destroy dynamic resources. That said, it is certainly possible to design your resource with a "destruct parameter" such that PUTs to it would cause the sketch to delete the resource. Most of the time, we anticipate that the thing represented by the resource will be sustained in the web for the benefit of many client sketches or apps.

Create the dynamic resource in your sketch

Sketch coding:

```
107  * Create an event resource that Chariot connects to your web of things.
108  *   --also set up the PUT callback that will receive RESTful API calls.
109  */
110 static String trigger = "event/tmp275-c/trigger";
111 static String attr = "title=\"Trigger\?get|obs|put\"";
112 static String eventVal = "{\"ID\":\"Trigger\",\"Triggered\":\"No\",\"State\":\"Off\"}";
113
114 bool triggerCreate()
115 {
116   if ((eventHandle = ChariotEP.createResource(trigger, 63, attr)) >= 0) {
117     if (ChariotEP.triggerResourceEvent(eventHandle, eventVal, true)) {     // set its initial condition (JSON)
118       ChariotEP.setPutHandler(eventHandle, triggerPutCallback);           // set RESTful PUT handler
119     } else {
120       SerialMon.println(F("Error creating trigger!"));
121       return false;
122     }
123     return true;
124   }
125
126   SerialMon.println(F("\nCould not create resource!"));
127   return false;
128 }
```

Lines 110-112 provide a resource name, an attribute string (clients see both by doing GETs on the ".well-known/core" resource). Line 112 shows the format of the event value that will be sent by CoAP to all subscribed OBServers for this resource.

Line 116 calls the library to create the resource with a buffer of 63B.

Line 117 will cause the string, `eventVal`, to be saved as the new resource value. The value 'true' tells Chariot to signal any observers with the new value stored, which will be unlikely. The reason to do this is that it will also force the resource to be initialized in an unarmed or triggered state, waiting for a rearming by the sketch.

Line 118 sets up a PUT handler (callback) for taking action on PUT requests arriving from remote sketches and apps.

Note that the handle for this resource is stored in an int named `eventHandle`.

PUT API for this sketch

Chariot's RESTful API includes the functionality to use the PUT primitive to set any parameters your sketch has created and recognizes as mutable from the web. This is done by registering a PUT callback in the sketch creating the resource. Subsequently, CoAP PUT URLs that include the syntax "&param=<name>&val=<value to set>" can be processed. Parameters for this example are the temperature value to trigger event notifications at, a calibration value (in Celsius) to compensate for the fact that the TMP275 onboard temp sensor is actually measuring Chariot's temp, the function to be applied ('>' or '<' in this case), and the state of the trigger ('on' and 'off'). In order to receive trigger events published by Chariot, you also need to issue an OBSERVE request to the DNS name (in this ex., chariot.c350f.local) identifying the mote hosting the resource creating sketch. Here are examples of PUT commands and their responses. The names of the parameters are at the discretion of the sketch author. In this experiment, the resource has been named "event/tmp275-c/trigger."

Set trigger to 35C:

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "event/tmp275-c/trigger",
       opts = "param=triggerval&val=35";

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=triggerval&val=35
```

Response:

```
chariot.c350f.local: 2.05 CONTENT triggerval now set to 35
```

Set the calibration offset to -2C which will be added to the trigger value before testing to see if trigger conditions were met:

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "event/tmp275-c/trigger",
       opts = "param=caloffset&val=-2;
```

```
ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=caloffset&val=-2
```

Response:

```
chariot.c350f.local: 2.05 CONTENT caloffset now set to -2
```

Set the function to "Greater Than." This directs the owning sketch to detect the temp sensor to rising above 35C. When this happens it triggers and a "greater than 35C" event is distributed to registered subscribers.

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "event/tmp275-c/trigger",
       opts = "param=func&val=gt;

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=func&val=gt
```

Response:

```
chariot.c350f.local: 2.05 CONTENT func now set to gt
```

Check Chariot's temp sensor before enabling the trigger

This shows that the temperature sensor that Chariot comes equipped with can be accessed in different ways by creating new resources and assigning semantics and functions in your sketch. In setting up our dynamic trigger, to check the temp, we need only refer to the resource that comes as part of standard Chariot firmware (see Table 1).

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "sensors/tmp275-c",
       opts = "";

ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/sensors/tmp275-c?get
```

Response:

```
chariot.c350f.local: 2.05 CONTENT 29.0(C)
```

You may wish to further instrument the example by setting a concurrent periodic observation of Chariot's temp sensor, which is what the sketch is monitoring from the arduino. The default period for the sensor is 7 seconds. This can be changed by URL, as in this example changing it to 10 seconds.

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "sensors/tmp275-c",
       opts = "period=10&units=secs";

ChariotEP.coapRequest (COAP_POST, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/sensors/tmp275-c?post&period=10&units=secs
```

Response:

```
chariot.c350f.local: 2.04 CHANGED Period set to 10 seconds
```

This turns on the ten-second periodic observation of the temp sensor responses:

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "sensors/tmp275-c",
       opts = "";

ChariotEP.coapRequest (COAP_OBSERVE, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/sensors/tmp275-c?obs
```

Response:

```
chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 23.4,"Unit":"C"}
chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 23.4,"Unit":"C"}
```

```
... @ 10sec intervals
chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 23.4,"Unit":"C"}
```

Make a final check by 'fetching' trigger parameters:

This example sketch's put handler creates a put function called fetch that will return the current parameter values of the trigger resource.

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "event/tmp275-c/trigger",
       opts = "param=fetch&val=noise";

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=fetch&val=noise
```

Response:

```
chariot.c350f.local: 2.05 CONTENT {"ID":"Trigger","State":"Off", "Func":"GT",
"TriggerVal":35.00,"CalOffset":-2.00}
```

Turn it on and cause a trigger:

First issue a request to observe the dynamic resource—causing event publications to arrive at your client sketch or app. Note that we will now be observing the same sensor in two different ways and as two different resources: "sensors/tmp275-c" is sending readings every 10 secs and observing "event/tmp275-c/trigger" awaits the temperature to rise above the trigger value set at 35C:

Sketch API:

```
String mote = "chariot.c350f.local",
       resource = "event/tmp275-c/trigger",
       opts = "";

ChariotEP.coapRequest (COAP_OBSERVE, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?obs
```

Response:

```
chariot.c350f.local: 2.05 CONTENT event/tmp275-
c/trigger={"ID":"Trigger","Triggered":"Yes","State":"Off"} //note that sketch formats this in JSON
```

Now arm the trigger by setting the parameter “state” to “on”:

Sketch API:

```
String mote = “chariot.c350f.local”,
       resource = “event/tmp275-c/trigger”,
       opts = “param=state&val=on”;

ChariotEP.coapRequest (COAP_PUT, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=state&val=on
```

Response:

```
chariot.c350f.local: 2.05 CONTENT state now set to on
```

Heat up the temperature sensor and observe the trigger event:

Now use a heat gun or hair dryer to cause the trigger. The subscription in the above observe step will result in a notification to be published to subscribers.

The trigger is parameterized in the sketch as shown above. Applying heat and watching the temperature every 10 seconds will enable us to wait expectantly for the trigger. This will also be observable if you have set up a wireshark capture. With the trigger set to 35C and our calibration offset at -2C, we should see an event while passing a 33C sense value:

Trigger Response:

```
chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 33.3,"Unit":"C"}

chariot.c350f.local: 2.04 CHANGED event/tmp275-
c/trigger={"ID":"Trigger","Triggered":"Yes","State":"Off"}
```

Note that the sketch turns the trigger off, requiring it to be rearmed much like a one-shot logic analyzer or oscilloscope analogue. This and any other custom event trigger behavior is completely controlled by the sketch author.

Searching your web-of-things for resources

Chariot provides capabilities enabling resource discovery. The CoAP standard, RFC 2752 and related, contain a resource listing request, "/.well-known/core" that is responded to with a list of name-attribute string pairs for the resources of devices running the CoAP protocol. Chariot always responds with the core resources (see Table 1) and any dynamic ones added by the sketch running on that mote.

Chariot also provides a special resource, "search/" that can be used to request a full or partial search of any Chariot mote for resources. The response to this request is a JSON formatted string that tells how many hits were returned, and the name of the first.

The Chariot API contains a helper function, `getMotes()`, that will return the list of DNS names currently visible to enable convenient mesh-wide resource searches.

Getting a list of all resources at a mote using CoAP's /.well-known/core resource (cf. RFC 7252)

Sketch API:

```
String mote = "chariot.c350e.local",
       resource = ".well-known/core",
       opts = "";

ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/.well-known/core?get
```

Response:

```
chariot.c350e.local: 2.05 CONTENT /.well-known/core
,/search;title="search";rt="search" ,/sensors/battery;title="Battery status";rt="Battery"
,/arduino/digital;title="Digital Pin: ?get|post|put|observe|delete &pin=2..13 &val=1|0"
,/arduino/analog;title="Analog Pin: ?get|post|put|observe|delete &pin=0..5 &val=0..1024"
,/arduino/mode;title="Pin Mode: ?get|put&pin=0..13&val=input|input_pullup|output"
,/sensors/tmp275-c;title="Chariot TMP275 sensor"?get|{obs|del}|{put|post&period=1..59&units=
secs|mins}
,/sensors/radio;title="Chariot RADIO"?get|{obs|del}|{put|post&period=1..59&units=secs|mins}
,/sensors/fxos8700cq-c/accel;title="Chariot FXOS8700cq 3D-
accel"?get|{obs|del}|{put|post&period=1..59&units=secs|mins}
,/sensors/fxos8700cq-c/mag;t itle="Chariot FXOS8700CQ 3D-
mag"?get|{obs|del}|{put|post&period=1..59&units=secs|mins}
,/event/tmp275-c/trigger;title="Trigger?get|obs|put"
```

Searching a mote for an instance of a particular resource

Chariot has a special resource, "search", that returns full and partial matches of strings applied against the list of resources of a particular mote. It also returns the number of matches for the argument applied. For example, if we wanted to know if this node was running the example sketch that creates a networked temperature trigger and we knew its trigger resource name, we could use this fully specified search:

Sketch API:

```
String mote = "chariot.c350e.local",
       resource = "search",
       opts = "name=event/tmp275-c/trigger"; // dynamic resource names must begin with "/event";

ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/search?get&name=event/tmp275-c/trigger
```

Response:

```
chariot.c350e.local: 2.05 CONTENT {"Name": "tmp275-c/trigger", "Matched":"event/tmp275-c/trigger",
"Which": 1, "Hits": 1}
```

This was an exact or complete search—with the full name of resource specified. The number of "hits" returned is one.

Partial resource searches of motes

If in our sketch we, or a client app, wish to find out which resources in /sensors are currently available on this node, we can request a more general search:

Sketch API:

```
String mote = "chariot.c350e.local",
       resource = "sensors", // dynamic resource names must begin with "/event"
       opts = "";

ChariotEP.coapRequest (COAP_GET, mote, resource, TEXT_PLAIN, opts, response);
```

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/search?get&name=sensors
```

The answer returned shows that there are a number of 'hits' matching sensors, the first of which is the battery sensor:

Response:

```
chariot.c350e.local: 2.05 CONTENT {"Name": "sensors", "Matched":"sensors/battery", "Which": 1,
"Hits": 5}
```

We can ask for the other hits by extending the search URL to include "which" hit we're interested in seeing, for example the second instance:

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c350e.local/search?get&name=sensors&which=2
```

Response:

```
chariot.c350e.local: 2.05 CONTENT {"Name": "sensors", "Matched":"sensors/tmp275-c", "Which": 2,
"Hits": 5}
```

It is possible to perform more "educated" partial searches. For example, if temperature sensors conform to the convention "tmp" plus a descriptive tag in their naming, then we can find them quite easily:

Sketch API or URI from webapp to Chariot:

```
coap://chariot.c351d.local/search?get&name=sensors/tmp
```

Response:

```
chariot.c351d.local: 2.05 CONTENT {"Name": "sensors/tmp","Matched":"sensors/tmp275-c","Which":
1,"Hits": 1}
```

And so on. Note that search responses are returned as JSON strings. Resources, such as the standard sensors can return JSON responses as shown above.

Interfacing the Chariot mesh to Cloud Services

It is possible to connect Chariot low-power meshes to cloud services with very little effort. Blynk (blynk.cc) provides a very interesting drag-n-drop app builder for Android and iPhone smartphones. We have provided an example that connects through the Blynk cloud to an Android app. Our example sketch runs on the WeMos D1 R2 ESP8266 WiFi/UNO-compatible board with Chariot. Since these accesses varying according to service, please see our examples folder for sketch and README. We have other examples that connect to cloud services (e.g., google sheets) using Temboo's solution that will be coming to our examples gallery soon.

Local Chariot status and configuration commands

There are a number of Chariot commands that are available for submission through the Serial window of your sketch. These commands are provided to help builders of arduino webs-of-things to ascertain and change important conditions and parameters that determine performance. When operational parameters are changed, their new values are stored in permanent memory locations. You must reset Chariot for these new values to take effect.

"**health**" – This command will provide you with the DNS name of your Chariot and tell you where the operational Chariot console feed can be obtained. This can be displayed on your computer with the use of a 3V TTL USB cable connected to programs such as Tera Term.

```
2.05 CONTENT chariot.c3515.local is active. Details on UART#0 @57600bps
```

"**motes**" – This command tells you who your neighbors are on the wireless mesh your Chariot is connected to. Like "health" this command also displays the name of your Chariot:

```
2.05 CONTENT chariot.c3515.local neighbor motes:
chariot.c350f.local
chariot.c350b.local
```

Sensors – providing the names "**radio**", "**temp**", "**accel**" displays the current values of your Chariot's onboard sensors. These are useful for measuring wireless performance when positioning your Chariot or experimenting with antennas. You can see the temperature of your chariot when performing experiments, and determining movement.

```
2.05 CONTENT Lqi: 255, Rssi: -29(dBm)
2.05 CONTENT 22.5(C
2.05 CONTENT aX:139 aY:-177 aZ:2129 mX:59 mY:-310 mZ:-265
```

"**chan**" or "**chan=<new channel to set>**" – This command displays and changes the IEEE 802.15.4 channel used by this Chariot. Legal channel numbers are 11 through 26, and correspond with different frequencies, in a way analogous to Wi-Fi. Chariot is shipped using channel 26. If you need to change to a different channel, for example, 18, enter the following command: **chan=18.** You should receive a response like this:

```
2.05 CONTENT RF channel stored: 18.
```

To simply retrieve the current RF channel number, simply type "**chan**" to receive:

```
2.05 CONTENT RF channel: 26
```

See the datasheet on the Atmel ATMega256RFR2 Wireless MCU or any online reference on IEEE 802.15.4/Zigbee for frequencies these correspond to.

"**txpwr**" or **"txpwr=<new setting>"** – This command controls the amount of transmit power used by Chariot's radio. It is set to '0' when Chariot is shipped. This value corresponds to the maximum transmit power levels. It can be set to values between 0 (max. power) and 15(min. power). See the datasheet on the Atmel ATMega256RFR2 Wireless MCU for the power levels (in dBm) these correspond to. To lower the TX power level by about half, the command, **txpwr=8** could be used:

```
2.05 CONTENT Transmit power stored: 8.
```

"**panid**" or "**panid=<new PANID>**" — This command changes the identity of the Personal Area Network scanned by Chariot. Operations occur "intra-PAN", meaning that arriving and departing traffic stay within the PAN, except for that sent to the broadcast PAN ID (0xFFFF). Those frames are received by all PANs. Retrieve your PANID using "**panid**":

```
2.05 CONTENT PANID: 0xABCD
```

This is the "default" PAN ID for 802.15.4 networks. This is a 16-bit hexadecimal number. It must contain 1-4 hexadecimal digits prepended with "0x". To modify it for operation on a PAN with a different ID (ex, 0x1234), use the following: "**panid 0x1234**" which returns,

```
2.05 CONTENT PANID stored: 0x1234.
```

"**panaddr**" or "**panaddr=<new PANADDR>**" — This command changes the identity of Chariot known to your network. This value is also synonymous to the serial number of your Chariot. Retrieve your PANADDR using "**panaddr**":

```
2.05 CONTENT PANADDR: 0x3515
```

This is a 16-bit hexadecimal number. It must contain 1-4 hexadecimal digits prepended with "0x". If you wish to modify your PANADDR (aka short MAC address) for operation of your Chariot with a different ID (ex, 0x2686), use the following: "**panaddr 0x2686**" which returns,

```
2.05 CONTENT PANADDR stored: 0x2686.
```

Note regarding usage of commands that change values (in eeprom)

Some important considerations apply when modifying the operational parameters of your Chariot. First and foremost, it is possible to render Chariot incapable of communicating with the others in your network. This is usually due to a value of the RF channel ("chan") that has become different from the rest of the motes. It can also happen with incompatible (i.e., different) values of the PANID ("panid") or PANADDR("panaddr") from those of the rest of your network. This is why the commands described above are administered through the local serial interface of your sketch. They can always be queried and modified..

Changes that have been made to RF channel, TX power, PANID, or PANADDR parameters require a Chariot reset to take effect. This can be done on most Arduinos from your sketch by closing the serial window (if open) and clicking the "Serial Monitor" button in the upper right-hand corner of the Arduino IDE. You can also press the reset button on your Arduino or Chariot. Note your configuration is displayed upon Chariot's serial monitor when it is restarted.

Using Chariot's serial monitor

Chariot has two hardware serial channels. One is used for communication of requests and responses with Arduino, one is set aside for Chariot monitoring. Chariot periodically displays its mDNS table, Arduino channel status, address, routes and neighbor tables. Also displayed are details regarding data exchanges between Arduino and Chariot.

We use GearMo USB to 3.3v TTL converter. These can be purchased at Amazon (we get ours there). See, for example, https://smile.amazon.com/gp/product/B004LBXO2A/ref=od_aui_detailpages00?ie=UTF8&psc=1.

UART 0, clearly labelled on Chariot has its TX pin jumpered to the RX pin of the converter cable (yellow lead), and Chariot's RX pin to the TX pin of the converter cable (orange pin). It is recommended that the cable's ground pin (black lead) be connected to any of the grounds available on Chariot.

Once your USB to TTL cable is hooked up you can set up your favorite USB terminal emulator to display information presented on the monitor. We use Tera Term here because we monitor from Windows systems. There are many to choose from. An example of our monitor window and it's setup @ 57600 baud rate is immediately below:

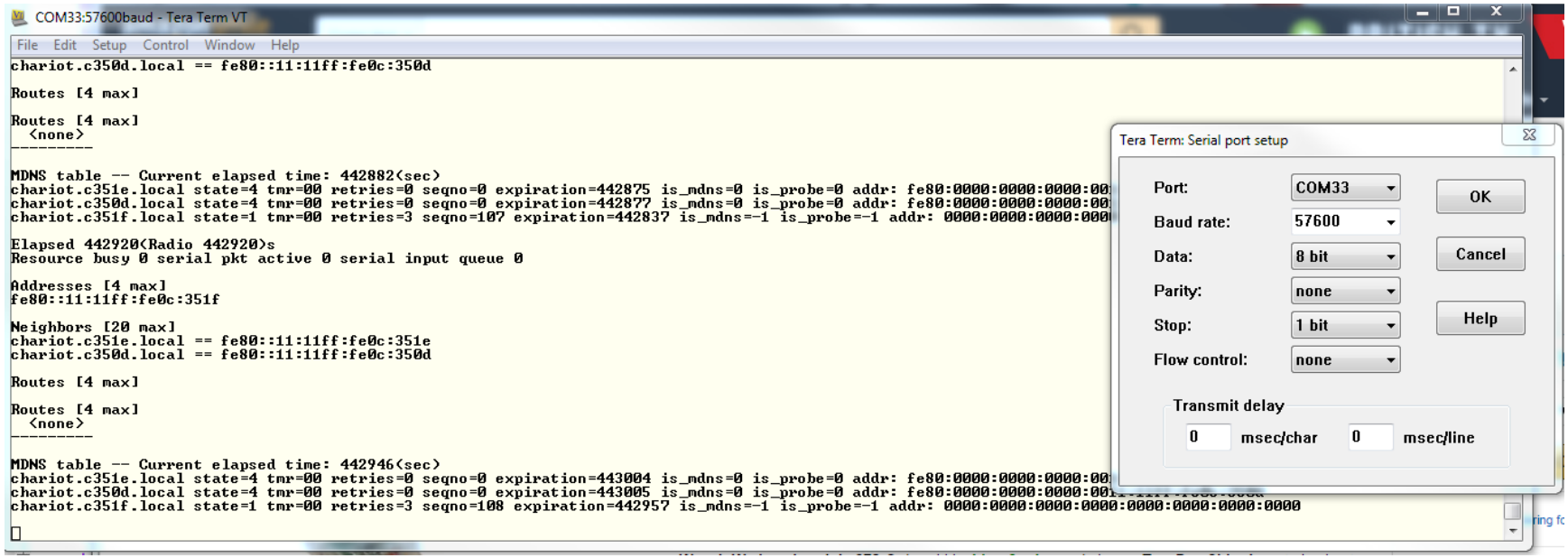

Figure 1. Tera Term display of Chariot monitor data also showing UART configuration.

Using Wireshark to observe mesh traffic interactions

Wireshark is a network packet analyzer. A network packet analyzer will try to capture network packets and tries to display that packet data as detailed as possible. Wireshark can be used to easily examine application and network data flows on a Chariot 6LoWPAN mesh, revealing exchanges at all layers, most importantly the CoAP application layer. This software is avalailable at wireshark.org and requires an IEEE 801.15.4 wireless interface in order to sniff packet traffic. Most often an usbstick that supports IEEE 802.15.4 and a Windows or linux computer are used to capture traffic in real-time and store it for later analysis. A number of suppliers sell these usb sticks. We use them from Atmel and from ubisys. Both are available on Amazon.

In order to capture traffic, you must know the channel currently in use by your Chariot mesh (default is 26 which doesn't overlap with WiFi) and set your usb stick to that channel. You must also set a capture filter to "zep". This tells Wireshark to limit itself to traffic originating and

terminating on the mesh. That's pretty much all you need to know to get results like the ones we got below. Wireshark also provides all documentation online and their terrific website (wireshark.org).

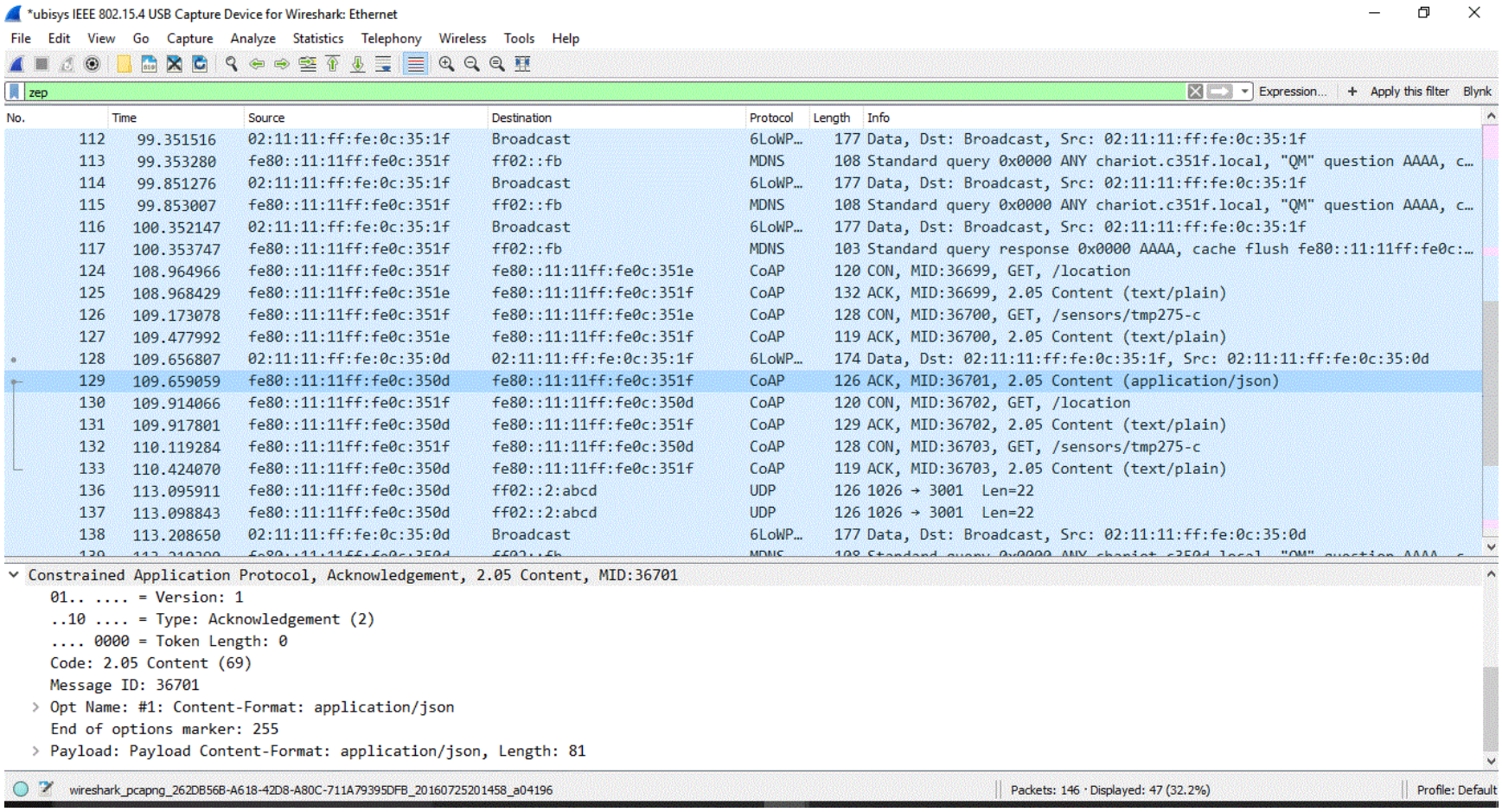

Figure 2. Capture of Chariot traffic using Wireshark and an IEEE 802.15.4 ubisys USB stick (see upper right hand corner).

Using Chrome and Firefox to experiment with URI construction and app development

For makers and developers it is often useful to have simple tools to create and examine web-of-things application scenarios. The frontend we provide is a very simple JavaScript interface that sets up and connects websockets to a server running on an Arduino running the server. We provide an ethernet sample in the examples folder. The figure below depicts an exchange featuring a resource search.

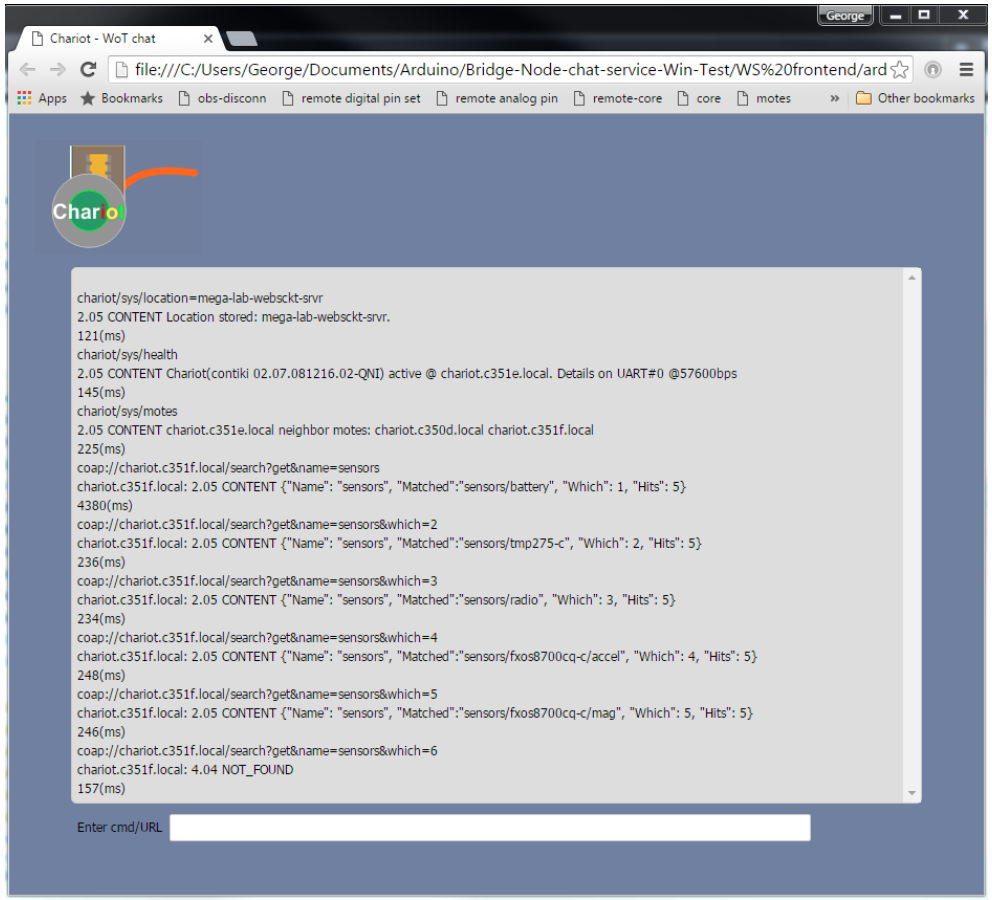

Figure 3. Chariot websocket browser interface for Chrome.

A brief introduction to Arduino web-of-things applications using

Smart things are designed and built to work on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

- **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery.
- **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. CoAP introduces a new RESTful primitive OBSERVE that allows web-apps subscribe to events published by web-of-things components, such as those using Chariot.
- **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control
- **Stateful interactions through hyperlinks**: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

Arduino 6LoWPAN CoAP web-of-things network using Chariot

The following diagram depicts a 6LoWPAN (IEEE 802.15.4) network in which a distributed Arduino application is constructed. One of the Arduinos (rightmost) could be an ESP8266 Wi-Fi shield, Yun, Ethernet shield equipped Mega, etc., that has access to the internet In the diagram:

EP—endpoint. This is a Chariot-equipped Arduino having control over a number of physical resources, all of which are accessed through

RD—resource directory. The node containing this functionality is able to access all resources of the EP's residing in the wireless network.

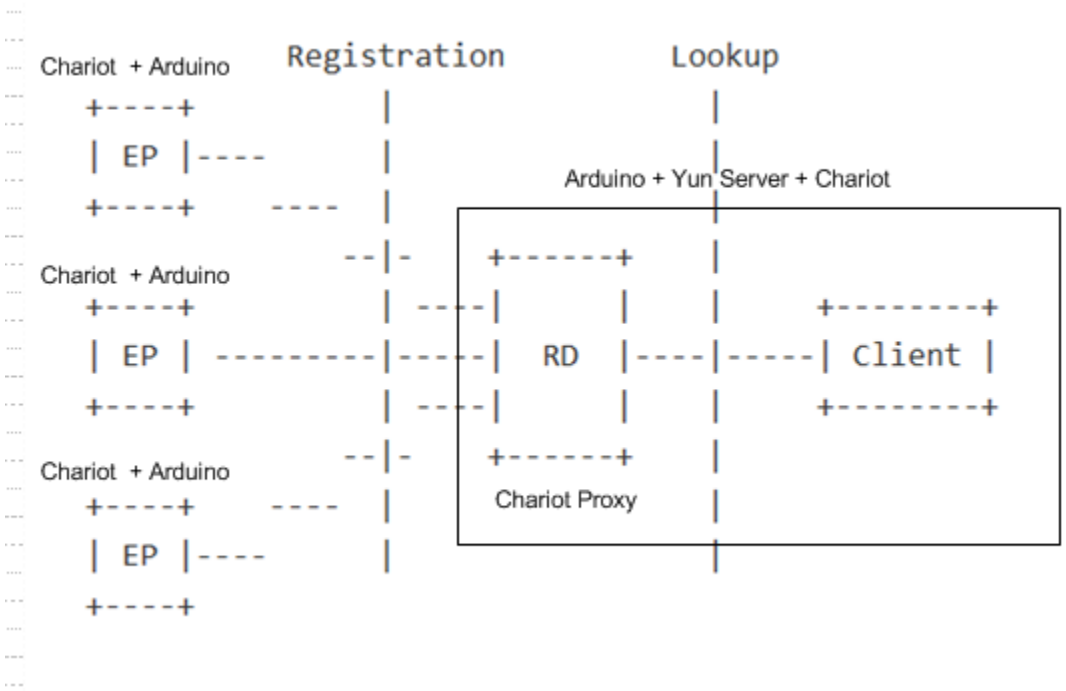

Figure 3. RESTful model of resources in the Chariot web-of-things architecture—RD == 'Resource Directory'

Chariot Web proxy model

A Reverse Websocket-CoAP Proxy (HC proxy in RFC 7252) is accessed by clients (web browsers, webapps, etc.) supporting websockets, and handles their requests by mapping and/or tunneling these to CoAP requests which are forwarded to CoAP servers, subsequently mapping back received CoAP responses to responses. This mechanism is transparent to the client, which may assume that it is communicating with the intended target server. In other words, the client accesses the proxy as a request target.

See Figure 2 (below) for an example deployment scenario. Here an HC Proxy is placed server-side (Yun), at the boundary of the Constrained Network domain, to avoid any web traffic on the Constrained Network and to avoid any (unsecured) CoAP multicast

traffic outside the Constrained Network. The DNS server is used by the client to resolve the IP address of the HC Proxy and optionally also by the HC Proxy to resolve IP addresses of CoAP servers.

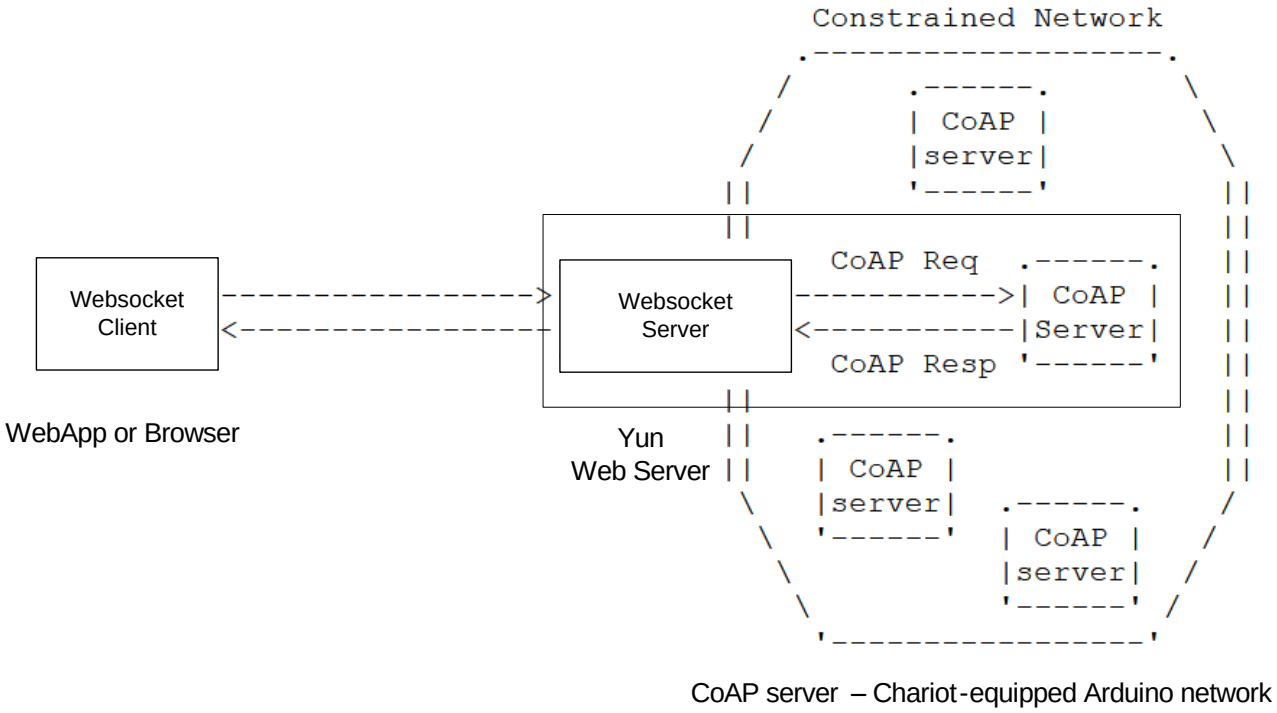

Figure 4.Example Chariot CoAP deployment. The standalone objects labelled “CoAP server” consist of Chariot-equipped Arduinos. Alternative websocket servers based upon the inexpensive ESP8266 WiFi Arduinos (e.g., WeMos D1 UNO-compatible) and W5100-compatible ethernet shields using the websocket library are also readily available. See examples folder.