Qualia networks incorporated

Chariot: Web-of-Things Shield for Arduino

URL examples from a browser or web-app websocket connection

Getting and setting Arduino analog and digital pin values from the web

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 an analog pin

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

```
Arduino @ 18:14: chariot.c350f.local: 2.05 CONTENT Pin A5 set to 358
```

Get current value for an digital pin

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

```
Arduino @ 18:14: chariot.c350f.local: 2.05 CONTENT Pin D12 set to 0
```

Make digital pin an output

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

```
Arduino @ 10:41: chariot.c350f.local: 2.05 CONTENT Pin D12 configured as OUTPUT
```

Set a new value to a digital pin in OUTPUT mode.

```
coap://chariot.c3507.local/arduino/digital?put&pin=12&val=1
```

```
Arduino @ 10:42: chariot.c350f.local: 2.05 CONTENT Pin D12 set to 1
```

Test it by resetting to its original value (or hooking up a led)

```
coap://chariot.c3507.local/arduino/digital?put&pin=12&val=0
Arduino @ 10:43: chariot.c350f.local: 2.05 CONTENT Pin D12 set to 0
```

Pin sharing between Arduino and Chariot

Arduino and Chariot use the serial port for the exchange of commands and information. On UNO, 2 digital pins, 11 and 12, are allocated for this. On Leonardo (including Yun), they are digitals 4 and 11. Note that Leonardo is no longer supported and will be dropped from Qualia support as well. All boards share 3 gpio's that consume digital's 7-9 and one external interrupt pin which uses digital pin 3 (not currently in use by Chariot). Other boards, such as Mega 2560 have more hardware serial ports that remove the requirement for using two of the digital gpio's for the serial channel. In summary, digital pins 7-9 are used for gpio sharing and control between Chariot and Arduino. For UNO, pins 11 and 12 are also unavailable as they are used for transmit and receive serial channels between Chariot and Arduino.

Chariot's builtin sensors

Chariot motes come standard with high performance temperature sensor and a 6-axis accelerometer. 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, as needed. All of the sensors will output JSON strings if the parameter string ";ct=50" is added to the URL. See the Radio sensor.

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 one the I2C, no gpio's are consumed for their support. Reading the value is done from an app or neighbor using the URL scheme shown below in the sketch example:

```
coap://chariot.c350f.local/sensors/tmp275-c?get
Arduino @ 12:18: 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 may convert to any other units, as requirements call for.

Onboard Accelerometer

The accelerometer is a Freescale Semiconductor 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 and 25Hz with both accelerometer and magnetometer active. This device appears as separate accelerometer and magnetometer resources to external accessors. Both support get, obs, and del RESTful operations and can be programmed, as with the temp sensor, to perform periodic reporting of their current values:

/sensors/fxos8700cq-c/accel:

```
coap://chariot.c350e.local/sensors/fxos8700cq-c/accel?get;ct=50
chariot.c350e.local: 2.05 CONTENT {"ID":"FXOS8700CQ Accel","X":597,"Y":24,"Z":-4635}
```

and:

/sensors/fxos8700cq-c/mag:

```
coap://chariot.c350e.local/sensors/fxos8700cq-c/mag?get;ct=50
chariot.c350e.local: 2.05 CONTENT {"ID":"FXOS8700CQ Magnetometer","X":14,"Y":342,"Z":32}
```

Both Sensors

These sensors provide many functions beyond the RESTful API. As with the temperature sketch example, these may be created in your sketch and exported over the web. The example sketch, *webofthings_event_trigger*, gives an example of this using the temperature sensor. Both are completely controllable from the sketch, which is free to provide URL semantics for using the application functionality via GET 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 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:

```
coap://chariot.c350f.local/sensors/radio?get
Arduino @ 18:20: chariot.c350f.local: 2.05 CONTENT Lqi=255 Rssi=-44dBm
```

We can ask for the values to be returned as JSON by including its type (“ct=50”):

```
coap://chariot.c350e.local/sensors/radio?get;ct=50
chariot.c350e.local: 2.05 CONTENT {"ID":"ATMega256RFR2 Radio","Lqi":255, "Rssi":-45dBm}
```

Of the two, RSSI relates primarily to 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 2.550V to 3.675V:

```
coap://chariot.c350e.local/sensors/battery?get;ct=50
chariot.c350e.local: 2.05 CONTENT {"ID":"Chariot VCC Reading","Val": 3.525,"Unit":"V"}
```

Searching the mesh for resources

Chariot has a special resource that returns 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 request all resources from `chariot.c350e.local` in our network, using the RFC7252 standard request: `coap://chariot.c350e.local/.well-known/core?get`

```
chariot.c350e.local: 2.05 CONTENT /.well-known/core;ct=40
,/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"
```

IF we merely wanted to know if this node was running the example sketch that creates a networked temperature trigger, we could use the resource search: `coap://chariot.c350e.local/search?get&name=tmp275-c/trigger`

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

This was precise search—with the type of sensor designated. The number of "hits" returned is one. If we, or our software agent, wished to find out which sensors are currently available on this node, we can request a more general search:

```
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:

```
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:

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

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

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.

Set up a web-accessible trigger event from your sketch

Example sketch: "*Chariot_EP_sketch_webofthings_event_trigger*"

The URLs in these examples were issued from a Chrome browser running our html and JavaScript websocket frontend. They are being interpreted on a connected arduino Yun by a simple node.js backend relay service which is passing URLs and commands over the Yun-arduino "bridge" where the input is delivered to Chariot. Chariot's CoAP proxy prepares the request and, if the DNS address is found to be on the 6LoWPAN wireless cloud, forwards it. Note 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, manipulate, and destroy dynamic resources.

Using PUT for setting or changing parameters in your sketch from the web

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. Subsequently, CoAP PUT URLs that include the syntax "&param=<name>&val=<value to set>" can be processed by the sketch. Parameters for this example are the temperature value to trigger 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 Chariot shield with DNS name chariot.c350f.local that is plugged into the Arduino that is hosting the 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."

PUT API for this sketch

1.) Set the trigger to occur at a temperature of 35C:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=triggerval&val=35
Arduino @ 12:12: chariot.c350f.local: 2.05 CONTENT triggerval now set to 35
```

2.) Set the calibration offset to -2C which will be added to the trigger value before testing for an event

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=caloffset&val=-2
Arduino @ 12:14: chariot.c350f.local: 2.05 CONTENT caloffset now set to -2
```

3.) Set the function to "Greater Than."

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=func&val=gt
Arduino @ 12:15: chariot.c350f.local: 2.05 CONTENT func now set to gt
```

4.) Enable the trigger. Make certain that you have an OBSERVE active for this resource prior to this if there is a chance of triggering.

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=state&val=on
Arduino @ 12:16: chariot.c350f.local: 2.05 CONTENT state now set to on
```

Check Chariot's temp sensor before enabling the trigger

```
coap://chariot.c350f.local/sensors/tmp275-c?get
Arduino @ 12:18: chariot.c350f.local: 2.05 CONTENT 29.0(C)
```

You may wish to further instrument the example by setting 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 10 seconds.

```
coap://chariot.c350f.local/sensors/tmp275-c?post&period=10&units=secs
Arduino @ 12:19: chariot.c350f.local: 2.04 CHANGED Period set to 10 seconds
```

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

```
coap://chariot.c350f.local/sensors/tmp275-c?obs
Arduino @ 12:31: chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 23.4,"Unit":"C"}
```

Make a final check by 'fetching' trigger parameters used by the sketch:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=fetch&val=noise
Arduino @ 12:55: 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 ask to observe it—causing event publications to arrive at your browser window:

```
coap://chariot.c350f.local/event/tmp275-c/trigger?obs
Arduino @ 12:56: chariot.c350f.local: 2.05 CONTENT event/tmp275-
c/trigger={"ID":"Trigger","Triggered":"Yes","State":"Off"}
```

Second turn the trigger state to "on":

```
coap://chariot.c350f.local/event/tmp275-c/trigger?put&param=state&val=on
Arduino @ 12:56: 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 your websocket browser window. This is done by:

```
coap://chariot.c3507.local/sensors/tmp275-c?obs
Arduino @ 12:57: chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 23.8,"Unit":"C"}
```

The trigger is parameterized in the sketch as shown above. Applying heat and watching the Chariot websocket window on our browser that is monitoring 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 33C:

```
Arduino @ 13:20: chariot.c350f.local: 2.05 CONTENT {"ID":"Chariot TMP275","Val": 33.3,"Unit":"C"}

Arduino @ 13:20: 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.

Finally, you can terminate the periodic temp sensor observation like this.

```
coap://chariot.c350f.local/sensors/tmp275-c?del
Arduino @ 13:30: chariot.c350f.local: 2.02 DELETED Chariot TMP275 observation terminated
```

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.

It is recommended that when changing values, it is done *in absentia* from the network.

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 from your sketch by 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.

Introduction to Arduino web-of-things applications using Chariot RESTful web services

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) is either a Yun or an Ethernet shield equipped Arduino that has access to the internet (IPv4 is presumed—connection not shown). 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.

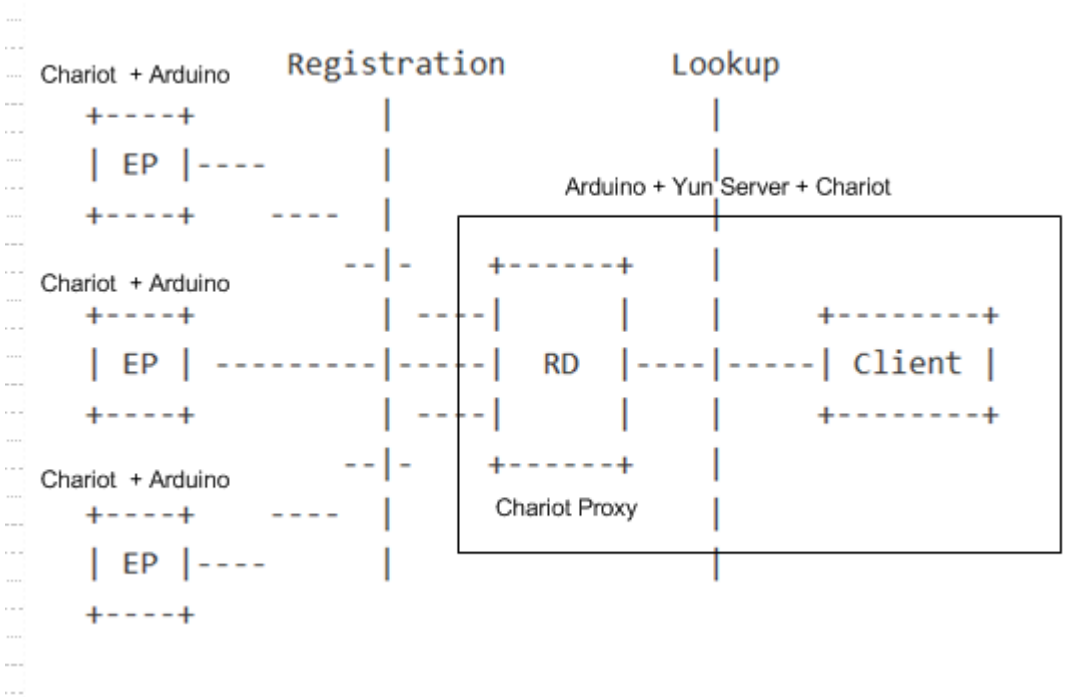

Figure 1. RESTful model of resources in the Chariot web-of-things architecture—RD == 'Resource Directory'

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.

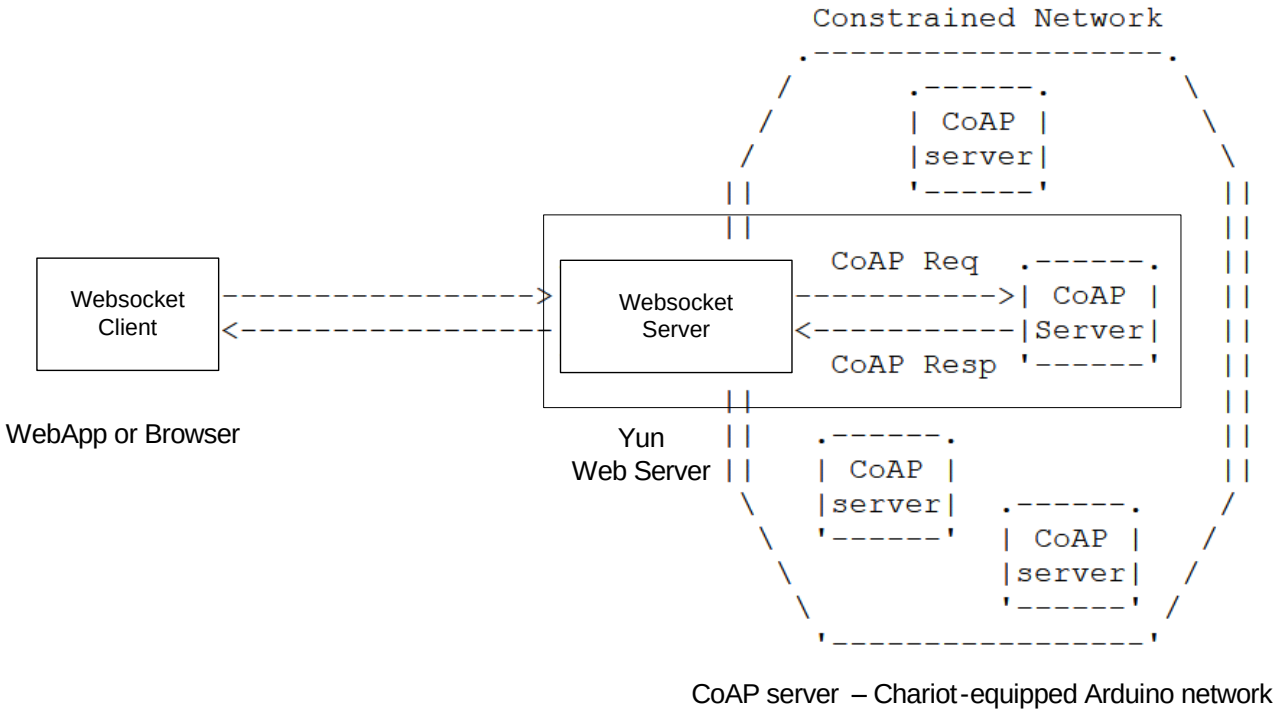

Figure 2. Chariot CoAP deployment mode. The standalone objects labelled “CoAP server” consist of Chariot-equipped Arduinos

WebApp access to resources: URI construction

The Arduino Yun includes a web server that, in addition to handling web pages stored on a micro SD card, is capable of processing URI’s that are specific to the Arduino side of the board (see http://www.arduino.cc/en/Main/ArduinoBoardYun?from=Products.ArduinoYUN for a system-level description of the Yun).

Table 1. Functional mapping using the Yun Bridge

URI	Resulting Action (Arduino libraries)
arduino/digital/13	digitalRead(13)

arduino/digital/13/1	digitalWrite(13, HIGH)
arduino/analog/2/123	analogWrite(2, 123)
arduino/analog/2	analogRead(2)
arduino/mode/13/input	pinMode(13, INPUT)
arduino/mode/13/output	pinMode(13, OUTPUT)

The portion of the URI, "/arduino" causes the Yun webserver to direct the link to the arduino side from processing, over the "Bridge". Our URIs will all contain the "/arduino" substring, plus much more in certain cases (see below).

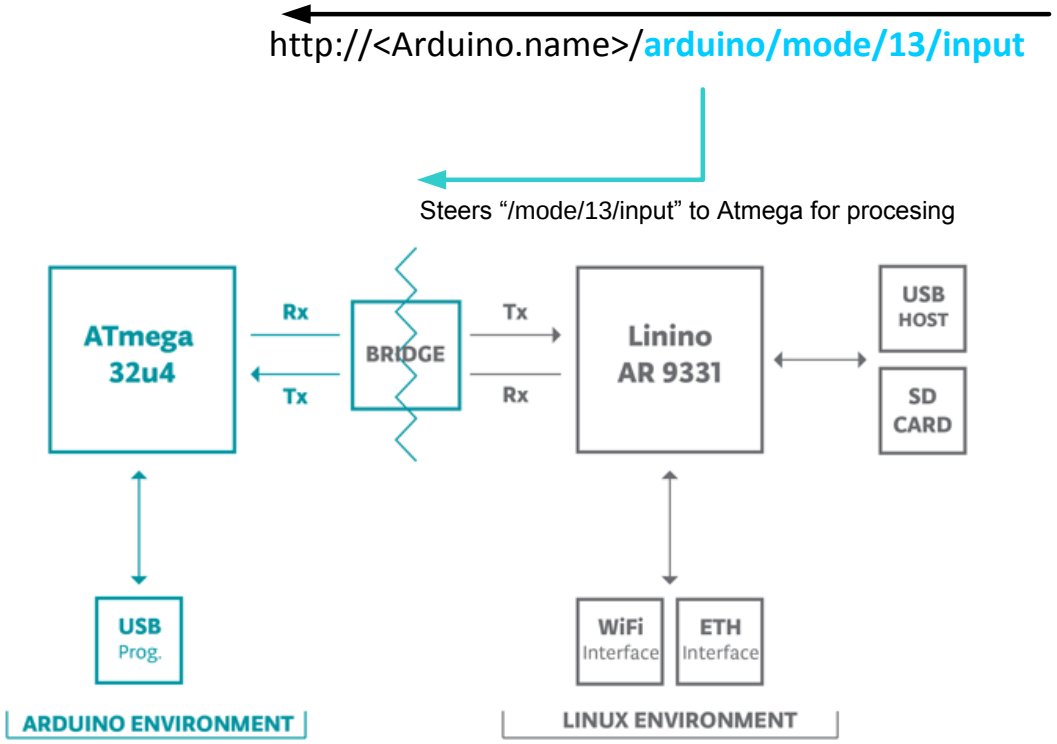

Figure 3. Arduino Yún system diagram and processing flow for URI's containing "/arduino".

Chariot-based Web-of-Things model

As extended by Chariot, the Arduino model above takes on the following structure:

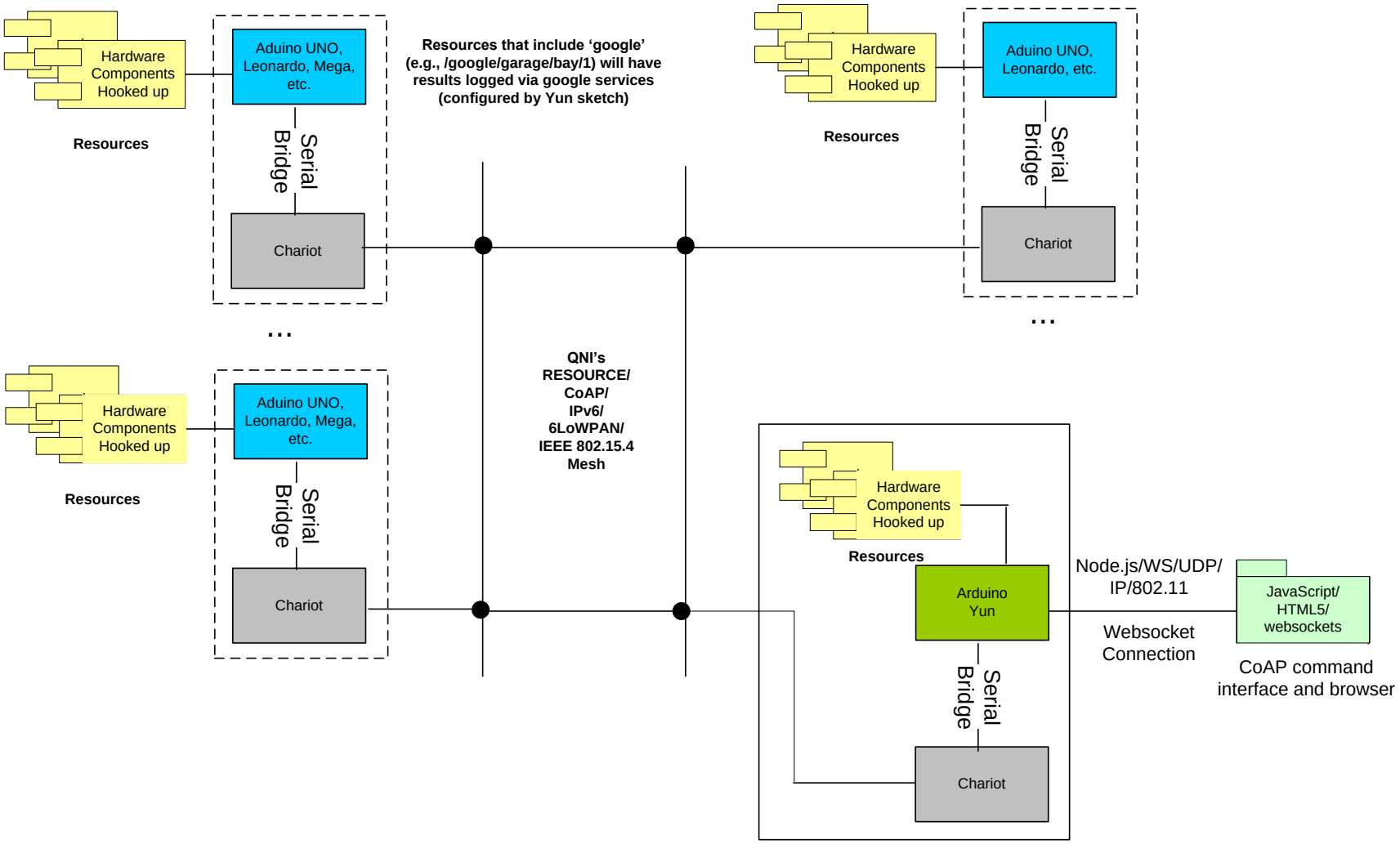

Figure 4. Chariot extension of Arduino IoT to web-of-things model.