Integration Guide for 3rd Parties (DRAFT)

Target audience: *technical* third parties looking to integrate eTactica hardware into external software packages.

Topics covered

- MQTT data streams available on each gateway
- Configuring MQTT bridges OUT to your own MQTT broker
- Scripting to reformat data
- ?? Ask for more topics

Topics **not** covered (here at least).

- Integration of EB/EM directly into Modbus systems

Overview of gateway operation

A brief overview of how the gateway operates will help to understand the type of data available, and what it means. The gateway runs a Modbus acquisition program (*mlifter*) which continuously polls all configured devices (both local and remote) and posts all the live readings directly to the onboard MQTT broker. Each device is polled nominally every 2 seconds, and the variables collected from each device can vary immensely, see the examples below. (Note that these are not the *only* variables available from EM/EB, just what is commonly collected)

Separate processes then listen to these MQTT streams and process them for sending to external software services, in whatever format they require. These can be as simple as directing the raw stream out, or selecting certain values, compressing to only send changed values, sending only every 15 minutes, almost anything you can think of. A web UI, (*channel monitor*) on the gateway allows monitoring all of this raw data.

	EM	**EB (power sync*)**	**Third party modbus device**
Measurements per point	voltage, power factor, current	current, voltage*, cos phi*, kwh import*	arbitrary
Measurements per device	net kwh, sum varh, frequency, temperature	temperature, frequency*	arbitrary

MQTT Streams available as data sources

Two major streams are available to use as your data source. (at present)

1. The SenML stream PUBLIC - status/local/json/device/# - Live reading data
2. The "simple" text stream PUBLIC - status/local/text/device/#

These two streams have pros and cons. The SenML stream provides a single rich JSON message for each Modbus device read. It includes all readings from that device, the timestamp it was read, information on the device itself, and includes error information if the modbus request failed.

The "simple" stream simply provides a topic for *every* reading, and the messages are just the data value itself, with no extra formatting. (Assume locale=C wrt to numeric formatting, ie, period for decimal separator, no thousands grouping indicator)

	SenML	**"simple"**
Pros	single reading, all information together timestamped full device information.	no parsing required easy to select single value of interest easy to visualise
Cons	harder to select single values more verbose, some redundant information	*many* more messages for some devices (can be much more CPU usage if you listen to entire tree) need to wait for multiple messages to correlate readings from the same device no timestamps in individual messages partial device information (on other topics in the stream) No unit provided, must be inferred from type name

Additionally, an MQTT stream is available that contains the user entered Cabinet Model. This contains information entered by the user such as cabinet groupings along with names, sizes and phase assignments of electrical breakers. This is optional information but it can make for much richer data. See PUBLIC - status/local/json/cabinet/<deviceid> Cabinet models

Methods of Integration

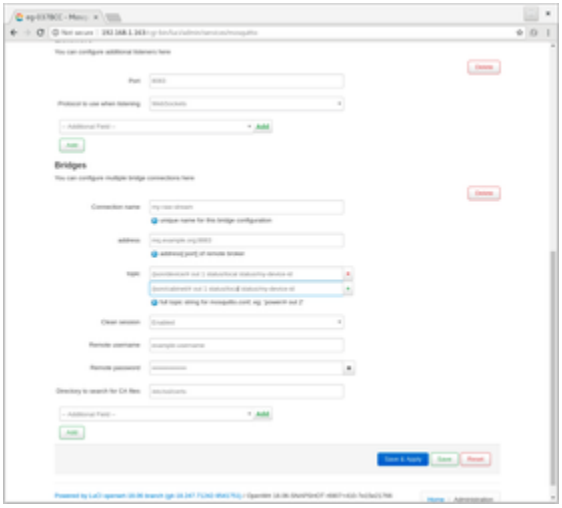
Integration methods fall largely into a couple of camps

- MQTT bridge OUT to your service, with or without reformatting of messages. (most common)
- MQTT bridge IN *from* your service, (reformatting/restructuring done at your end)
- MQTT->HTTP post from the gateway.

MQTT Bridge OUT to your service (without modifications)

This is the possibly the simplest configuration. The gateway is simply configured to send a copy of one it's internal data streams (see above) directly to the MQTT broker of your choice. That could be in your own local network, cloud hosted, or anything in between.

In it's simplest case, for this configuration, you can simply manually reconfigure the onboard MQTT broker, using the web UI or editing the config files directly. An example of this, where you send the local status/local/json/device/# stream to your own MQTT broker, using TLSv1.2, and remapping the topic to include a device id is shown below.



The direct config section corresponding to this is shown below.

/etc/config/mosquitto manual bridge configuration

You have the full flexibility of mosquittos' bridge configurations here. See https://mosquitto.org/man/mosquitto-conf-5.html for full details. This is both a blessing and a curse.

You may wish to simply bridge the topics directly, but provide an easier to use UI for your users to enable your service. The screenshot below shows the current simple style for this:

This requires *some* packaging of UI elements, and deploying this virtual "package" to your gateways, but it's straight forward and you can copy and paste your way from an existing example such as: https://github.com/remakeelectric/owrt_pub_feeds/tree/master/output-activitystream

This package includes the UI for user config and basic on/off, default settings, and also includes a stub example of running an extra daily service using the gateways built in cron support.

MQTT bridge OUT to your service (with reformatting or filtering)

This is the most common integration so far. A software service runs on the gateway, listens to one of the live streams, and handles reformatting/aggregating or any other transformation desired, and republishes the messages back to the broker. To simplify the software development, the onboard MQTT broker is configured to do the outbound bridging, so it handles reconnecting and security settings via standard configurations, and you write your software in the simple worldview of a local broker.

For example, the service for a third party cloud software package "99clouds" would simply publish anything they like to `ext/99clouds/out/<topic for remote broker>` and can subscribe to the topic `ext/99clouds/in/#` to receive any messages from that cloud service. This makes it easier to have multiple third party services co-exist, but if you have complicated needs, you can always manage your own outbound MQTT connections directly from your application.

Pros and Cons

Pros

- Packaged service, included and managed directly on each gateway.
- Service instance only has to consider one gateway's worth of data
- Send only what you need, when you need it.

Cons

- Must develop in a language suitable for deployment on the gateway.
- Must deploy your package to your gateways. (Can't use the factory image directly)
- A lot of boilerplate to handle packaging and monitoring

Language choices

We recommend Lua for most applications. It is easier to write, test and deploy than a C/C++ application, and importantly, has enough performance and low enough overhead that you can run applications easily on the gateway. You *can* use languages like python, but you will become memory and cpu constrained quite quickly, and we reallllly don't recommend it. If you want to use C/C++, see the "Using the SDK" document, and OpenWrt's documentation on "Building your own software" You can plausibly write simple applications even in just raw shell scripting, but you'll probably run into performance problems fairly quickly. The version of Lua on the gateways is Lua 5.1 (with the "lnum" patch applied, which is normally transparent)

Lua libraries available

This is not an exhaustive list, and many more can be enabled or added using the SDK

Library	Description
penlight	general utility library
lua-mosquitto	lua bindings to the mosquitto MQTT client library
cjson	high performance json encoded/decoder
socket + luasec	For making HTTP/HTTPS requests
posix	Allows access to common system calls
lfs	lua file system access
remake.ugly_log	Simple logging to console/syslog
lua bitop	Bitwise operations
lua zlib	lua bindings to zlib, for various compression tasks

Writing your application

Probably the best overview is to simply copy an existing application. "output-thingsboard" https://github.com/remakeelectric/owrt_pub_feeds/tree/master/output-thingsboard is a nice well contained application that demonstates:

- lua application handling reformatting output and processing inputs. See files/usr/bin/output-thingsboard.lua
- init script with configuration loading and MQTT bridge configuration See files/etc/init.d/output-thingsboard
- process monitoring see files/etc/monit.d/output-thingsboard.process.check
- Default configuration see files/etc/uci-defaults
- Web interface for enabling and basic paramter settings, see luarsrc/

In the application itself, there's a lot of boilerplate for verifying data formats and making mqtt connections and handling json decoding. You can largely copy this as is.

Init script

You have a lot of flexibility here. Normally, you should try and check whether your service is enabled or not, and ensure that any required MQTT connections are in place. You can choose to load and validate configuration here, check external scripts, validate network connectivity, the sky's the limit. A good example, with a lot of detail, but not too overwhelming is available at: https://github.com/remakeelectric/owrt_pub_feeds/blob/master/output-thingsboard/files/etc/init.d/output-thingsboard

Specific notes to follow there. The "add_instance" method is the core, handling loading configuration from the OpenWrt standard configuration files in /etc/config/output-thingsboard, enabling/disabling process monitoring, and starting the service itself. The other important section is how the MQTT bridges are torn down and recreated, along with topic remappings. Carefully using topic remappings makes sure that you don't interfere with any other software service. The general guideline has been to use the *ext/<service>/in* and *ext/<service>/out* trees for your service locally.

Process monitoring

The gateway has built in process monitoring via Monit and you can leverage this for your own processes. Simply include a snippet for your application, much as the example applications.

```
check process some-name matching '/usr/bin/lua /usr/bin/your-process'
# quite flexible here, see Monit documentation
 start program = "/etc/init.d/your-init-script start"
 stop program = "/etc/init.d/your-init-script stop"
        if totalmemory is greater than 40% for 3 cycles then restart
        if totalcpu is greater than 40% for 3 cycles then restart
 group your-group # group is optional too
```

Feel free to use as much or as little of monit as you like. The basic config is enough for most people

Packaging your service

Using the SDK to create your own applications Has initial steps you'll need. Primarily, you need to follow the Makefile example from output-thingsboard, or similar. This is only required if you want to share and install your package on multiple gateways. You can also simply copy files directly to the gateways manually using SCP, fetch them via HTTP(S) from the gateways using small scripts, or even use the toolchain to simply manually compile binaries by hand.

MQTT bridge IN from your service

As the eTactica gateways run an onboard MQTT broker, you can simply connect to this broker from your own service. You will need to manage the network access to this yourself. The plain unencrypted MQTT port 1883 is closed to remote access by default, but can be enabled in the web UI. If necessary, you can set up the full range of TLS security options, using the MQTT brokers built in configuration. The broker is mosquitto, and the webui allows configuration of *almost* all the settings mosquitto allows. Please let us know if there are settings you are not able to manage. Managing mosquitto is out of scope of this document.

This method can make a lot of sense if you don't feel comfortable writing software to run on the gateways directly, don't want to manage deployments to the gateways, and particularly if you will be deploying your software in the same site, so you don't have the same network security concerns as a cloud service.

This method also allows you to write your software in *any* language, using any environment you desire. You simply need MQTT client software, and to process and handle the data streams of your choice.

Examples

Some trivial examples that might give you a starting point are provided at https://github.com/remakeelectric/integration-examples

Simply searching for "MQTT" and your favourite programming language will give you *many* examples and ideas.

Lua

All of the existing service daemons for the gateway can be run as MQTT IN just as well as OUT, by simply changing what topics and broker addresses are used. This is actually how they are normally developed.

Python

Projects like *mqttwarn* offer a lot of flexibility out of the box. See https://github.com/jpmens/mqttwarn

MQTTHTTP post from the gateway

Not covered at this point. This is largely the same as MQTT bridge out, but your software service simply makes HTTP requests itself, rather than republishing your messages back to the MQTT broker.

See https://github.com/remakeelectric/owrt_pub_feeds/tree/master/output-dexma for an example service that does this.

PUBLIC - status/local/json/alert/{new,expired}

Introduction

The gateway monitors the live stream of readings and, in conjunction with the cabinet electrical model, monitors for overload conditions on all breakers. The alert is posted as a json message on MQTT topics, free for further integration, via MQTT bridges, or listening to these messages directly.

Availability

Version	Availability
2.10 or later	**AVAILABLE**

Configuration

The alerting engine works directly from the electrical cabinet model. If breaker sizes have not been set, the alerting engine cannot operate. No further configuration is currently necessary. Alerts are raised when the current on a circuit reaches 100% of nominal, and have maintained this state for more than 10 seconds.

MQTT Topics

When an alert condition is detected, and the hold time has been met, a message is published to `status/local/json/alert/new`. When the alert condition passes, and the hold time has been met, a message is published to `status/local/json/alert/expired` with the same sequence id. An aggregate state message is also published on changes, see status/local/json/alert/state documentation

JSON Root Elements

Field	Type	Description
label	string	User entered label for this breaker
sequence	numeric	the number in this particular sequence. Alerts are repeated with a backoff interval, this helps you identify new/old alerts
sequenceid	string	UUID identifying this particular alert event. Both new and expiry messages will use the same sequenceid
version	numeric	always 3
type	string	always "current"
nominal_limit	numeric	the threshold used that caused this breaker to register an alert

timestart_ms	numeric	timestamp in milliseconds since the epoch when this alert first went over threshold
timeend_ms	numeric	timestamp in milliseconds since the epoch when this alert first went under threshold. Only valid in expiry messages
last_reading	array of breaker reading elements	objects indicating the most recent received values for the configured breaker.

JSON breaker reading elements

Field	Type	Description
deviceid	string	represents the device that this breaker reading is from
reading	numeric	channel number this reading is from. In conjunction with deviceid, uniquely identifies the reading point
value	numeric	value at last reading
timestamp_ms	numeric	timestamp in milliseconds since the epoch when the last reading was collected

Examples

New, multiphase breaker.

```
{
    "last_reading": [
        {
            "deviceid": "4ECDD765A62E",
            "value": 28.15,
            "timestamp_ms": 1540392509445,
            "reading": 3
        },
        {
            "deviceid": "4ECDD765A62E",
            "value": 28.45,
            "timestamp_ms": 1540392509445,
            "reading": 4
        },
        {
            "deviceid": "4ECDD765A62E",
            "value": 29.12,
            "timestamp_ms": 1540392509445,
            "reading": 5
        }
    ],
    "sequenceid": "a2c3cc5f-b9b8-440a-ab91-58708260f531",
    "sequence": 2,
    "label": "Some breaker label",
    "version": 3,
    "timestart_ms": 1540392481360,
    "type": "current",
    "nominal_limit": 16
}
```


PUBLIC - status/local/json/interval/<X>min/<deviceid>/<readingkey>/<channel> - interval statistics

Availability

Gateway version	Status	
2.8.1 or earlier	NOT IMPLEMENTED	
2.10 or later	AVAILABLE	

Introduction

As part of the standard API, the gateway provides some aggregate interval statistics. For many integration purposes, the live streams are simply far too much data, and these interval topics allow picking and choosing what data you really want. No metadata about devices is provided here. The metadata subtopics of the live stream are largely static enough.

MQTT topics and variables

Four intervals are maintained concurrently. 1 minute, 5 minute, 15minute and hourly. You can mix and match the use of these topics in your own applications as you prefer.

Topic portion	description	
X	The interval. One of a fixed set of 1,5,15,60	
deviceid	The serial of the modbus device this data has been collected from	
readingkey	The same reading key as used in the live streams. See PUBLIC - status/local/text/device/# and PUBLIC - status/local/json/device/# - Live reading data for more information	
channel	The same optional sub channel as used in the live streams	

Message format

Each topic will contain a single json message, published retained, so that it is always available to new subscribers. All timestamps are in milliseconds since the linux epoch, as used throughout the gateway.

JSON Root Keys

Key	type	description
n	number	number of samples that contributed to this dataset
max	number	maximum value of datapoint in interval
max_ts	number	timestamp of max reading
min/min_ts		as max
mean	number	raw average of samples
stddev	number	raw stddev of all samples
first	number	raw value at start of interval
last	number	raw value at end of interval
ts_start	number	timestamp of start of interval
ts_end	number	timestamp of end of interval

Examples

Topic: status/local/json/interval/5min/ABCDFACECAFE/current/4

```
{
    "last": 40,
    "max": 42,
    "ts_start": "ts of start of window in millis",
    "first": 39,
    "min": 12,
    "mean": 30,
    "stddev": 3.3,
    "min_ts": "ts_in_millis",
    "ts_end": "ts of start of window in millis",
    "max_ts": "ts_in_millis",
    "n": 25
}
```

Device failing for entire window. We still publish an aggregate, but with 0 samples. If a device is *removed* from the configuration, you will simply stop receiving messages for that device.

```
{
    "ts_start": "ts of start of window in millis",
    "ts_end": "ts of start of window in millis",
    "n": 0
}
```

Bandwidth

Figure roughly 340 bytes per message or so. Will be more or less, given size of variables, and json encoding, but ballpark. Calculator attached that lets you choose interval in minutes, and number of bars/meters

PUBLIC - status/local/json/cabinet/<deviceid> Cabinet models

Availability

Gateway version	Cabinet Model format version	
2.2 or later (March 2018)	0.3	**AVAILABLE**
2 2.2	0.2	**DEPRECATED BUT COMPATIBLE**

Introduction

To ensure that third party data sinks have sufficient metadata to interpret live data streams, we added "cabinet editing" to the gate UI in v2. To make this responsive, we post each portion of the cabinet model as retained messages, so any consumer can subscribe and instantly receive the current model, and also be notified whenever the user changes them, without having to specifically request them. It's important to note that this metadata is only captured for *electrical* parameters. It does not attempt to allow arbitrary labelling of datapoints, only the primary metadata of breaker size and phase assignments that can only be captured by an on site technician.

MQTT Topics and retained flags.

The <deviceid> in the topic is the serial number of the device that this model document describes. Device IDs are unique within a gateway, but there is no guarantee of global uniqueness. All messages on these topics are retained, and given the importance of their metadata, these are even stored persistently even when no SD card is installed, on disk at /etc/remake.d/cabinet_model_cache. On bootup, these files are read from disk and republished for gateways that don't have mqtt persistence enabled via the SD card.

JSON Root Elements

Field	type	description
branches	array	Array of logical breaker objects
type	constant string "profile"	Used to help identify this message without having the topic attached.
cabinet	string	user provided cabinet name that all of these breakers belong to
version	number	presently, the value 0.3
validated	boolean	When present, means any required internal actions have been completed. (Such as assigning phases on power bars) You can normally ignore this field, the user provided metadata doesn't change.

JSON logical breaker objects

Field	type	description
ampsize	number	The user supplied numeric ampere size for this breaker
label	string	The user supplied string label of this breaker
points	array	an array of breaker reading point objects. Normally either 1 element or 3 elements long. Lists the physical reading points that make up this logical breaker.

JSON breaker reading point objects

These are simple tuples describing the phase and channel/reading point that make up a breaker. The mix of zero based and one based numbering is an unfortunate legacy. We're sorry about that.

Field	type	description
phase	number	One based phase assignment for this point, eg, 1, 2, 3
reading	number	Zero based channel index on this device, eg 0, 11

Examples version 0.3

General three phase meter

Topic: status/local/json/cabinet/110083A

```
{
    "deviceid": "110083A",
    "type": "profile",
    "version": 0.3,
    "branches": [
        {
            "points": [
                { "phase": 1, "reading": 0 },
                { "phase": 2, "reading": 1 },
                { "phase": 3, "reading": 2 }
            ],
            "ampsize": 16,
            "label": "KARLO1"
        }
    ],
    "validated": true,
    "cabinet": "other cabinet"
}
```

12 Pin eTactica Power Bar with mixed sizes and both single and three phase breakers

Topic: status/local/json/cabinet/2ACA65EBB19A

```
{
    "deviceid": "2ACA65EBB19A",
    "type": "profile",
    "version": 0.3,
    "branches": [
        {
            "points": [ {"phase": 1, "reading": 0 } ],
            "ampsize": 16,
            "label": "9a/1"
        },
        {
            "points": [ {"phase": 2, "reading": 1 } ],
            "ampsize": 16,
            "label": "9a/2"
        },
        {
            "points": [
                { "phase": 3, "reading": 2 },
```

```
                { "phase": 1, "reading": 3 },
                { "phase": 2, "reading": 4 }
            ],
            "ampsize": 63,
            "label": "BigTriple"
        },
        {
            "points": [ { "phase": 3, "reading": 5 } ],
            "ampsize": 16,
            "label": "9a/4"
        },
        {
            "points": [
                { "phase": 1, "reading": 6 },
                { "phase": 2, "reading": 7 },
                { "phase": 3, "reading": 8 }
            ],
            "ampsize": 8,
            "label": "little boy"
        },
        {
            "points": [ { "phase": 1, "reading": 9 } ],
            "ampsize": 16,
            "label": "9a/6"
        },
        {
            "points": [ { "phase": 2, "reading": 10 } ],
            "ampsize": 16,
            "label": "custom single"
        },
        {
            "points": [ { "phase": 3, "reading": 11 } ],
            "ampsize": 16,
            "label": "9a/8"
        }
    ],
    "validated": true,
    "cabinet": "my awesome cabinet"
```

```
}
```

Historical information

Examples Version 0.2

Version 0.2 is identical, but every single point entry includes the *deviceid* in addition to the phase and reading keys. This was a carryover from an earlier format that contained every single breaker in a single json document and allowed theoretically creating a three phase breaker across two different physical devices. This was never used in practice, and the complexity and redundance was dropped.

Example single phase "bar" style device

Example single phase "bar" style device

```
Topic: status/local/json/cabinet/2ACA65EBB19A

{
    "branches": [
        {
            "ampsize": 22,
            "label": "Q32.7",
            "points": [ { "phase": 1, "reading": 6, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.1",
            "points": [ { "phase": 1, "reading": 0, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.2",
            "points": [ { "phase": 2, "reading": 1, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.3",
            "points": [ { "phase": 3, "reading": 2, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.4",
```

```
            "points": [ { "phase": 1, "reading": 3, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "custom5",
            "points": [ { "phase": 2, "reading": 4, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.6",
            "points": [ { "phase": 3, "reading": 5, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.8",
            "points": [ { "phase": 2, "reading": 7, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.9",
            "points": [ { "phase": 3, "reading": 8, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.10",
            "points": [ { "phase": 1, "reading": 9, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.11",
            "points": [ { "phase": 2, "reading": 10, "deviceid":
"2ACA65EBB19A" } ]
        },
        {
            "ampsize": "22",
            "label": "Q32.12",
            "points": [ { "phase": 3, "reading": 11, "deviceid":
"2ACA65EBB19A" } ]
        }
    ],
    "cabinet": "mycab",
    "version": 0.2,
    "type": "profile"
}
```

PUBLIC - status/local/json/device/# - Live reading data

Availability

Gateway version		
version 2.0+	AVAILABLE	

Introduction

This stream is the "full" live stream, the original and primary method of integration. See also: PUBLIC - status/local/text/device/# Pros and cons are outlined on Integration Guide for 3rd Parties (DRAFT) A single message is published for every reading of a device, and the message is rich, with all information known at that time. There are two major portions, "senml" which covers the readings themselves, and "hwc" which is the "hardware configuration" information block, and covers information about device. As the "original" live stream format, there are some redundancies and legacy data here.

Topics and retained flags

No messages in this space are published as retained. Nominally, each device is read every two seconds, and each message contains all known information, so clients are simply expected to wait for the next message. Device IDs are provided by the device plugins, and are required to be unique within a gateway at least, but not necessarily globally unique.

Device	Topic
Succesfully probed	status/local/json/device/<deviceid>
Configured, but not yet probed	status/local/json/device

Message Format

At the root level, the following keys can be found

Key	Meaning	Example
timestamp_ms	milliseconds since unix epoch	1537443066583
deviceid	redundant copy from topic. Included so that messages can be completely standalone without further metadata	E643E5EE2391

senml	A SenML block, see below	
hwc	A HardWare Configuration block, see below	

SenML block

The SenML block is based on https://tools.ietf.org/html/draft-jennings-senml-10 Note that the SenML draft (it's still only a draft) has moved in different directions, and this is INCOMPATIBLE with the current draft: https://tools.ietf.org/html/draft-ietf-core-senml-13

The messages published by the gateway just use a simple subset of the functionality available. With reference to https://tools.ietf.org/html/draft-jennings-senml-10#section-6

SenML root elements	Usage on Gateway	Example
bn	always provided. Always "deviceid/" Note the trailing /, this allows direct concatenation with parameter entry names to form mqtt topics	E643E5EE2391/
bt	Always provided. This is the time of the reading, in **milliseconds** since the unix epoch. Note that the draft specified this in *seconds*, which we felt was too coarse.	1537443769351
bu	never provided	
ver	never provided	
e	always provided, with at least one element.	

SenML parameter entries	Usage on Gateway	Example
n	always provided. Predefined type names documented in PUBLIC - SENML name keys - registry of uses	frequency, current/4
v	always provided.	49.999
u	always provided.	Wh, Hz
t	never provided	
sv	never provided	
bv	never provided	
t	never provided (all readings are from the same time, see root level "bt"	
ut	never provided	

HWC block

The **H**ard**W**are **C**onfiguration block contains mostly static information about the device itself. It's quite redundant if you're exporting this out of the gateway, but makes for simple integration locally as each message has everything you need to know.

json key	json type	Meaning	example
slaveId	number	Decimal Modbus unit address	146
deviceid	string	Another copy of the device id	
lastPollTime	number	milliseconds since unix epoch when device was last read. Not *exactly* the same as the timestamp_ms, which is when this status message was created, but normally very close.	1537443084804
mbDevice	string	Modbus connection name	local
vendor	number	(optional) Numeric vendor code, provided by eTactica devices only.	21069

vendorName	string	String Vendor name	eTactica, Frer, Siemens
product	number	(optional) Numeric product code, provided by eTactica devices only.	18245
productName	string	Product name, as complete as device plugins can provide	"CE4DMID01", "EM21 Compatible"
pluginName	string	filename of the plugin in use for reading this device	etactica_eb-es.lua
pluginSource	string	whether a system or user plugin is being used	system or user
pluginCategory	string	Category of plugin. This only affects the dropdowns in the UI	electricity
typeOfMeasurementPoints	string	Fixed string, "generic" only for legacy compatibility.	generic

firmwareVersion	object	Provides a firmware version. Keys:
		"major" (number)
		"minor" (number)
		"dirty" (boolean) (Dirty is intended to indicate private software builds, you should never see this set to true!)
			<pre>{
			  "major": 4,
			  "minor": 12,
			  "dirty": false
			}</pre>

error	object	(optional) Only present if the device is having read problems. Keys:
		"status" (number) a numeric code, direct mapping to "meaning"
		"meaning" (string) a brief explanation of why the last reading failed.
		"failureCount" (number) number of failed readings in a row.
			<pre>{
			  "status": 3,
			  "meaning": "Modbus protocol",
			  "failureCount": 3
			},</pre>

Examples

Normal, successful readings

eTactica Current Bar - Topic status/local/json/device/6B768BF56888


```
{
    "timestamp_ms": 1446548186288,
    "deviceid": "6B768BF56888",
    "senml": {
        "e": [
            { "n": "current/1", "v": 0, "u": "A" },
            { "n": "current/2", "v": 0, "u": "A" },
            { "n": "current/3", "v": 0, "u": "A" },
            { "n": "current/4", "v": 0, "u": "A" },
            { "n": "current/5", "v": 0, "u": "A" },
            { "n": "current/6", "v": 0, "u": "A" },
            { "n": "current/7", "v": 0, "u": "A" },
            { "n": "current/8", "v": 0, "u": "A" },
            { "n": "current/9", "v": 0, "u": "A" },
            { "n": "temp", "v": 28.4, "u": "Cel" }
        ],
        "bn": "6B768BF56888/",
        "bt": 1446548186288
    },
    "hwc": {
        "slaveId": 136,
        "mbDevice": "local",
        "lastPollTime": 1446548186288,
        "deviceid": "6B768BF56888",
        "vendor": 21069,
        "product": 16964,
        "vendorName": "ReMake Electric",
        "pluginName": "etactica_eb-es.lua",
        "pluginSource": "system",
        "pluginCategory": "electricity",
        "typeOfMeasurementPoints": "generic",
        "numberOfMeasurementPoints": 9,
        "firmwareVersion": {
            "major": 3,
            "minor": 10,
            "dirty": false
        }
    }
}
```

eTactica EM (all mains meters) Topic: status/local/json/device/0004A3ED253F


```
{
    "timestamp_ms": 1513613581197,
    "deviceid": "0004A3ED253F",
    "senml": {
        "e": [
            { "n": "frequency", "v": 50, "u": "Hz" },
            { "n": "cumulative_wh", "v": 356152.35600000003, "u": "Wh"
},
            { "n": "cumulative_varh", "v": 42708.190999999999, "u":
"VArh" },
            { "n": "current/1", "v": 9.6660000000000004, "u": "A" },
            { "n": "volt/1", "v": 225.13999999999999, "u": "V" },
            { "n": "pf/1", "v": 0.96999999999999997 },
            { "n": "current/2", "v": 0, "u": "A" },
            { "n": "volt/2", "v": 226.33600000000001, "u": "V" },
            { "n": "pf/2", "v": 1 },
            { "n": "current/3", "v": 0, "u": "A" },
            { "n": "volt/3", "v": 229.49100000000001, "u": "V" },
            { "n": "pf/3", "v": 1 },
            { "n": "temp", "v": 32.420000000000002, "u": "Cel" }
        ],
        "bn": "0004A3ED253F/",
        "bt": 1513613581197
    },
    "hwc": {
        "slaveId": 63,
        "mbDevice": "local",
        "lastPollTime": 1513613581197,
        "deviceid": "0004A3ED253F",
        "vendor": 21069,
        "product": 18238,
        "vendorName": "eTactica",
        "pluginName": "etactica_em.lua",
        "pluginSource": "system",
        "pluginCategory": "electricity",
        "typeOfMeasurementPoints": "generic",
        "numberOfMeasurementPoints": 3,
        "firmwareVersion": {
            "major": 3,
            "minor": 20,
            "dirty": false
        }
    }
}
```

Failure / error messages

You can distinguish between failing unknown and failing known. Failing known will have a proper *deviceid* field, but will have the *error* block present. Failing unknown will have no deviceid in the topic, and no deviceid in the hwc block.

Failing, unknown device. Topic: status/local/json/device


```
{
 "timestamp_ms": 1537447210855,
 "message": "Failed to respond to 28 selected probes",
 "hwc": {
  "slaveId": 240,
  "mbDevice": "local",
  "lastPollTime": 1537447210353,
  "error": {
   "status": 6,
   "meaning": "Unrecognized",
   "failureCount": 1
  }
 }
}
```


Example failing, but known. (Might be permanent, might be intermittent)

Failing, but known device Topic: status/local/json/device/A4C2B38FC86D


```
{
    "timestamp_ms": 1513614058793,
    "hwc": {
        "slaveId": 109,
        "mbDevice": "local",
        "lastPollTime": 1513614058793,
        "error": {
            "status": 7,
            "meaning": "No SPI",
            "failureCount": 6378
        },
        "deviceid": "A4C2B38FC86D",
        "vendor": 21069,
        "product": 18245,
        "vendorName": "eTactica",
        "pluginName": "etactica_em.lua",
        "pluginSource": "system",
        "pluginCategory": "electricity",
        "typeOfMeasurementPoints": "generic",
        "numberOfMeasurementPoints": 3,
        "firmwareVersion": {
            "major": 3,
            "minor": 21,
            "dirty": false
        }
    }
}
```


Bandwidth

This message format is quite verbose, and the repeated publishing of the hwc (hardware config) blocks in *every* message means you probably **do n't** want to feed this stream directly to your cloud service. Attached is a little calculator to help you estimate data usage from the MQTT stream alone

As an example, a single mains meter, plus eight 12 channel power bars can generate around a GigaByte per DAY

PUBLIC - status/local/text/device/#

Availability

Gateway version		
less than 2.8	UNAVAILABLE	
2.8	OPTIONAL	Must be enabled manually with "uci set mlifter.remake.include_simple_text=1"
3+	AVAILABLE	On by default, but can be disabled.

Introduction

This stream is an alternative to the "full" live stream, offering an alternative method of integration. Pros and cons are outlined on Integration Guide for 3rd Parties (DRAFT)

For some integrations, particularly simple third parties and webhooks, accessing a single value on a single topic can be much simpler than extracting individual elements from the rich json object. Note, if you are planning on listening to the entire stream, the JSON stream is in almost all cases more cpu efficient.

MQTT device ids and retained flag

Devices that have been sucessfully probed will publish all values as retained, using the device id as their root topic element. This means that you will receive the latest values for all topics immediately upon connection, without having to wait for the next publish. Devices that have *not* been

probed successfully will *not* publish as retained, and will use a pseudo device id of `"unknown@<modbus connection id>:<modbus unit address>".` This allows you to follow devices still being probed, if desired, but doesn't pollute any retained topics with pseudo devices after they finish probing.

deviceid	This is the serial number of a modbus device. Where devices don't provide a serial number, device plugins provide a pseudo serial such as "productname-modbusaddress" These deviceids are unique within a gateway, but there's no particular guarantees on global uniqueness.
modbus connection id	eg "local", This is the "Connection name" value entered when configuring (rmeote) modbus devices
modbus unit address	as a decimal number, eg, 145

When data collection restarts, (after a configuration change, or device restart) **all** retained topics under status/local/text/device/# will be cleared by publishing a null message. This is to avoid dangling topics for devices that are no longer configured. Your application should be prepared to accept these null messages, and you can use them to track changes in config yourself.

Topics

Readings Topics

The general form for readings is `status/local/text/device/<deviceid>/<reading-key>/<channelnumber>`

This mirrors the basic topic structure of the json formatted live reading stream. See PUBLIC - status/local/json/device/# - Live reading data with the third position indicating the data format.

Readings are always published.

Defined reading-keys

Not all devices will or can provide all of these keys. But if they can, they should always use the same key. Precision will be as provided by the device being read, but units will always be consistent.

These are the *same* keys as used in the JSON formatted messages. See also PUBLIC - SENML name keys - registry of uses

reading-key	has channel number?	Unit	Meaning	Example
temp	optional	°C	without a channel number is Device internal temperature. Otherwise external channels	37.4
frequency	no	Hz	mains frequency	49.98
current	yes	A	Current on channel	12.5
volt	yes	V	Voltage on channel	229.4
pf	yes	-	power factor on channel. Defined that positive is import, negative is export. leading/lagging is ignored.	0.98
cumulative_wh	no	Wh	Sum of energy import on all channels less sum of energy export on all channels	6742129
cumulative_varh	no	varh	Sum of reactive import plus sum of reactive export	234122
wh_in	yes	Wh	imported energy for a specific channel.	12312.234
flownet	optional	m³	cumulative sum of flow	23432
flowrate	optional	m³/hr	rate of flow. period of integration not defined	2.45
voltage_supply	optional	V	voltage of power supply.	3.31
temp_flow	optional	°C	temperature of flowing fluid	85.4
pulse_count	yes	ticks	raw pulse count value	1234543

Metadata Topics

Metadata is published on a subtopic to make it a little easier to follow: `status/local/text/device/<deviceid>/meta/<meta-reading-key>`

Metadata is only published when it has changed, but is published retained, so you will receive all metadata when you first connect.

Standard Metadata

meta-reading-key	Meaning	example
timestamp_ms	unix epoch time (in milliseconds, ie, *1000) when this device was last read. Decode with `$ date -d @1537361330.704 => Wed Sep 19 12:48:50 GMT 2018`	`1537361330704`
mbAddress	Decimal Modbus unit address	146
mbDevice	Modbus connection name	local
vendor	String Vendor name	eTactica, Frer, Siemens
product	Product name, as complete as device plugins can provide	"CE4DMID01", "EM21 Compatible"
pluginName	filename of the plugin in use for reading this device	etactica_eb-es.lua
pluginSource	whether a system or user plugin is being used	system or user
pluginCategory	Category of plugin. This only affects the dropdowns in the UI	electricity
firmwareVersion	free text provided by the plugin.	4.12M, 1.0, 2016.12-abc
error	A (brief) textual explanation of why the last reading has failed. (Cleared when no error)	"No SPI", "Modbus protocol", etc
errcode	numeric code corresponding to the error string (Cleared when no error)	3, 7
failCount	number of failed readings in a row (Cleared when no error)	

Optional Metadata

meta-reading-key	Meaning	example
vendorCode	eTactica magic number	21069
productCode	eTactica magic number for this particular product. Can be used to identify hardware revisions	16975

Examples

eTactica Mains meter, full readings

```
$ mosquitto_sub -t "status/local/text/device/E643E5EE2391/#" -v -h
192.168.1.163
status/local/text/device/E643E5EE2391/meta/timestamp_ms 1537363413733
status/local/text/device/E643E5EE2391/meta/mbAddress 145
status/local/text/device/E643E5EE2391/meta/mbDevice local
status/local/text/device/E643E5EE2391/meta/vendorCode 21069
status/local/text/device/E643E5EE2391/meta/productCode 18245
status/local/text/device/E643E5EE2391/meta/vendor eTactica
status/local/text/device/E643E5EE2391/meta/pluginName etactica_em.lua
status/local/text/device/E643E5EE2391/meta/pluginSource system
status/local/text/device/E643E5EE2391/meta/pluginCategory electricity
status/local/text/device/E643E5EE2391/meta/firmwareVersion 4.12
status/local/text/device/E643E5EE2391/frequency 49.984
status/local/text/device/E643E5EE2391/cumulative_wh -43787.9
status/local/text/device/E643E5EE2391/cumulative_varh 67458.7
status/local/text/device/E643E5EE2391/current/1 0
status/local/text/device/E643E5EE2391/current/2 0
status/local/text/device/E643E5EE2391/current/3 0
status/local/text/device/E643E5EE2391/volt/1 237.258
status/local/text/device/E643E5EE2391/volt/2 237.794
status/local/text/device/E643E5EE2391/volt/3 239.575
status/local/text/device/E643E5EE2391/pf/1 0
status/local/text/device/E643E5EE2391/pf/2 0
status/local/text/device/E643E5EE2391/pf/3 0
status/local/text/device/E643E5EE2391/temp 34.98
^C
```

eTactica Mains meter, subsequent readings

```
status/local/text/device/E643E5EE2391/meta/timestamp_ms 1537363526707
status/local/text/device/E643E5EE2391/frequency 50.056
status/local/text/device/E643E5EE2391/cumulative_wh -43787.9
status/local/text/device/E643E5EE2391/cumulative_varh 67461.3
status/local/text/device/E643E5EE2391/current/1 0
status/local/text/device/E643E5EE2391/volt/1 237.684
status/local/text/device/E643E5EE2391/pf/1 0
status/local/text/device/E643E5EE2391/current/2 0
status/local/text/device/E643E5EE2391/volt/2 238.224
status/local/text/device/E643E5EE2391/pf/2 0
status/local/text/device/E643E5EE2391/current/3 0
status/local/text/device/E643E5EE2391/volt/3 240.009
status/local/text/device/E643E5EE2391/pf/3 0
status/local/text/device/E643E5EE2391/temp 35.11
```

Bandwidth

While this stream contains no repeated elements as in the JSON stream, it's still relatively high bandwidth. The bandwidth is primarily due to the rather long topic names. With the base topic plus the datapoint names, the topic + reading is around 110 bytes per metric. If you are bridging this topic outbound, you can strip off the leading portion, and save ~24 bytes per metric but it will always be rather voluminous. A rough calculator is attached below

PUBLIC - status/local/json/alert/state

Introduction

This is a continually updated summary topic, containing the current state of all alerts. See PUBLIC - status/local/json/alert/{new,expired} for the general overview of the alerts. This topic is generally used for status screens or general display, not as an integration point.

Availability

Version	Availability	
2.10 or later	AVAILABLE	

MQTT Topic

Whenever new information about the state of an active alert is received (typically, on every single device read, nominally every 2 seconds) `status/local/json/alert/state` receives a new retained message.

JSON Elements

The root object is a dictionary, where the keys are the nominal "breaker id" and the values are the same format as used in the new/expired messages. See PUBLIC - status/local/json/alert/{new,expired} for full details on this.

"breaker id" format

This is an internal unique identifier for a single/multi phase breaker. It is *only* used to make sure that the dictionary of alerts doesn't contain duplicates. The format is the device identifier, appended with the channel numbers that make up the breaker.

Example		Description
"2ACA65EBB19A-5"		Channel 5 on the device 2ACA65EBB19A
"2ACA65EBB19A-4-5-6"		Channels 4,5 and 6 on device 2ACA65EBB19A

Example messages

An example of presently active alerts on both a single and three phase breaker.

```
{
    "2ACA65EBB19A-4-5-6": {
        "last_reading": [
            {
                "deviceid": "2ACA65EBB19A",
                "value": 0,
                "timestamp_ms": 1544461054195,
                "reading": 4
            },
            {
                "deviceid": "2ACA65EBB19A",
                "value": 58.7,
                "timestamp_ms": 1544461054195,
                "reading": 5
            },
            {
                "deviceid": "2ACA65EBB19A",
                "value": 0,
                "timestamp_ms": 1544461054195,
                "reading": 6
            }
        ],
        "type": "current",
        "version": 3,
        "label": "breaker_label_from_cabmodel",
        "timestart_ms": 1544461040074,
        "nominal_limit": 16,
        "sequence": 2,
        "sequenceid": "38ffae95-924a-40b9-85d6-83510315557c"
    },
    "CAFEFACE0001-2": {
        "last_reading": [
            {
                "deviceid": "CAFEFACE0001",
                "value": 56.8,
                "timestamp_ms": 1544461031412,
                "reading": 2
            }
        ],
        "type": "current",
        "version": 3,
        "label": "label_from_cabinet_model",
        "timestart_ms": 1544455976195,
        "nominal_limit": 16,
        "sequence": 2,
        "sequenceid": "c1592268-f7b5-4f44-8c43-e5c51745470c"
    }
}
```

PUBLIC - SENML name keys - registry of uses

> Do *NOT* go making up new names here just as you see fit. The registry of names is how partners can interpret our data!

IN USE

Name	Standard Unit	Notes	Where
current/%d	A	Current (in Amps) on channel %d	EB/EM
volt/%d	V	Voltage (in volts) on channel %d	EM
pf/%d	pf (only to avoid bugs on ET2)	Power factor between -1 and 1, based on direction of active power. (No indication of leading/lagging)	EM. EB when reporting to Etactica2
pulse_count/%d		Raw ticks on an input. No interpretation	ER, 3rd party pulse meters
temp and temp/%d	Cel	Temperature. Could be on cpu temperature, or environmental.	EB2+, EM2+,
temp_internal	Cel	Internal device temperature	CS Instruments VA5xx plugin
wh_in/%d	Wh	Active import Watt hours	EB
cumulative_wh	Wh	Total Active Import - Total Active Export Watt hours (signed)	EM
cumulative_varh	Varh	Total Lagging Reactive + Total Leading Reactive	EM
frequency	Hz	Frequency of mains, not normally declared which phase, or whether average of all phases or not	EM, EB
flownet	m3	import - export consumption of a volume, normally water, analogous to cumulative_wh	Water meters, both ultrasonic and some OBIS meters, gas flow meters
flowrate	m3/h	flow rate, non specified integration time	Water meters, both ultrasonic and some OBIS meters, gas flow meters
velocity	m/s		CS Instruments VA5xx plugin
voltage_supply	V	power supply voltage, where measured	CS Instruments VA5xx plugin

PROPOSED / niche

Name	Standard Unit	Notes	Where
output/%d		binary state of a digital output	GC5 SFAR modules
input/%d		binary state of a digital input	GC5 SFAR modules
temp_gas	Cel	Temperature of measured flow **CAUTION** Will probably be renamed to temp_flow	CS Instruments VA5xx plugin

Connecting to Gateway and Simple Setup

This is a description on how to connect to the eTactica Gateway (EG) and how to do a simple setup where a Wizard will guide you through all the steps.

How to Get Connected to the eTactica Gateway

Most commonly, this is done by WiFi. By default every Gateway comes with an open WiFi interface (wireless hotspot) for initial configuration. The SSID for the wireless hotspot is always ***"eTactica eg-xxxxxx",*** where -xxxxxx is a unique number for each Gateway.

Alternatively you connect by using your Ethernet connection.

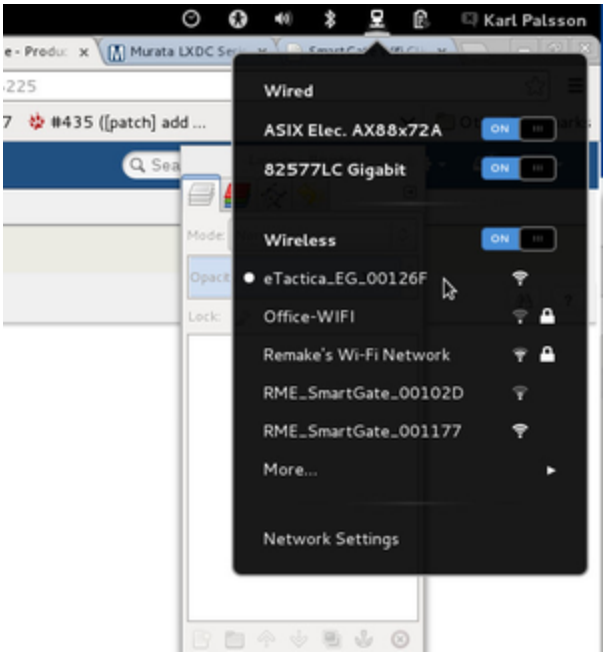
Connection via WiFi

Step 1 - Connect to WiFi hotspot

Use the normal operating system method for connecting to a new wireless hotspot.

On Windows it looks something like this:

and very similar on Linux:



Step 2 - Visit the admin console website

If you have connected via WiFi, the URL to the administration console is always http://192.168.49.1. Type this IP address into your web-browser to get access.

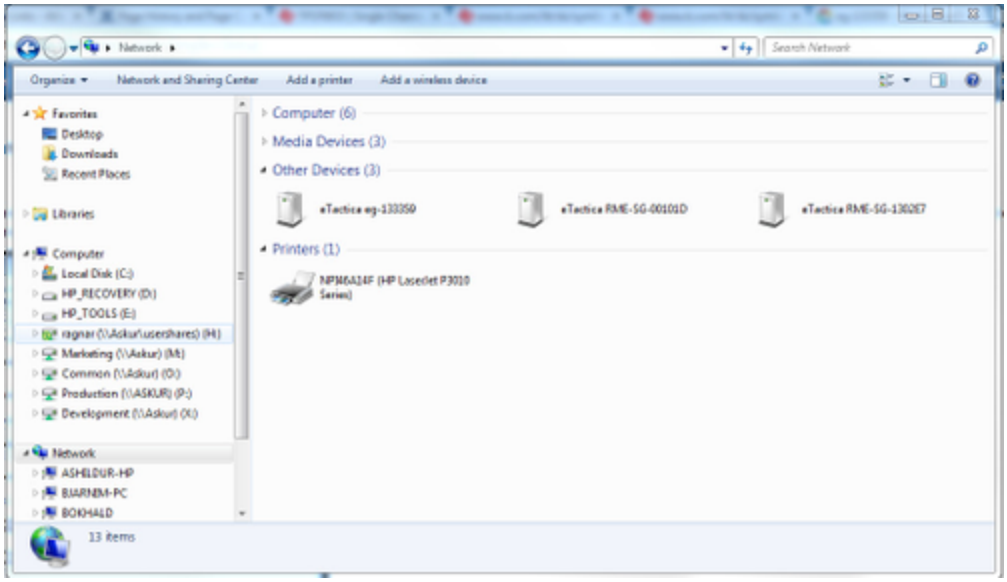
Connection via Ethernet

Windows

If the device has been connected to your existing Ethernet network, as is usual, you can find the device in *Windows Explorer -> Network -> Other devices*, as shown below.

Simply double click the name of the device you wish to connect to and you will automatically be directed to the admin console page of the gateway, via your web-browser.

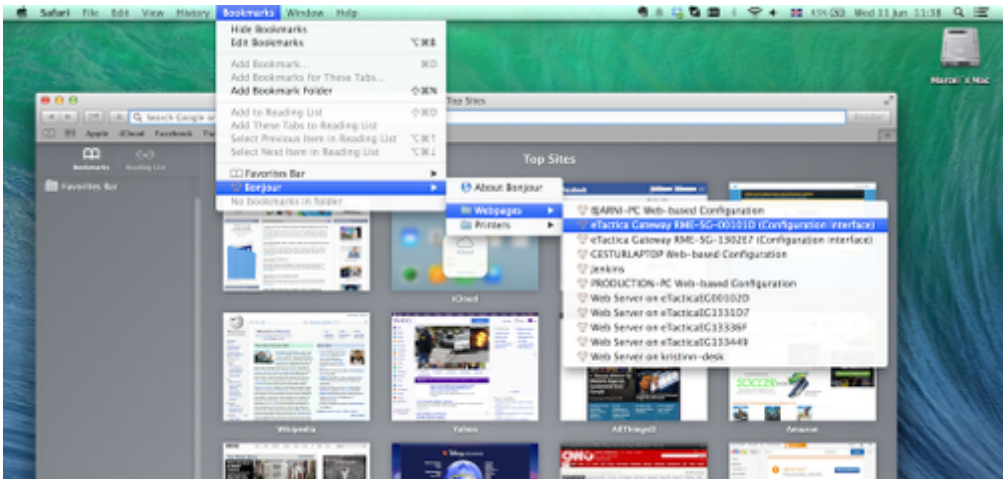
The name of the device shown here, will also match ***"eTactica eg-xxxxxx",*** where -xxxxxx is a unique number for each device.



OS X

On OS X, using the Safari Browser, you can visit "*Bookmarks->Bonjour Bookmarks*" and choose the entry for the matching device.

Note: you may need to enable browsing Bonjour Bookmarks first, see information at http://support.apple.com/kb/PH11848



Linux

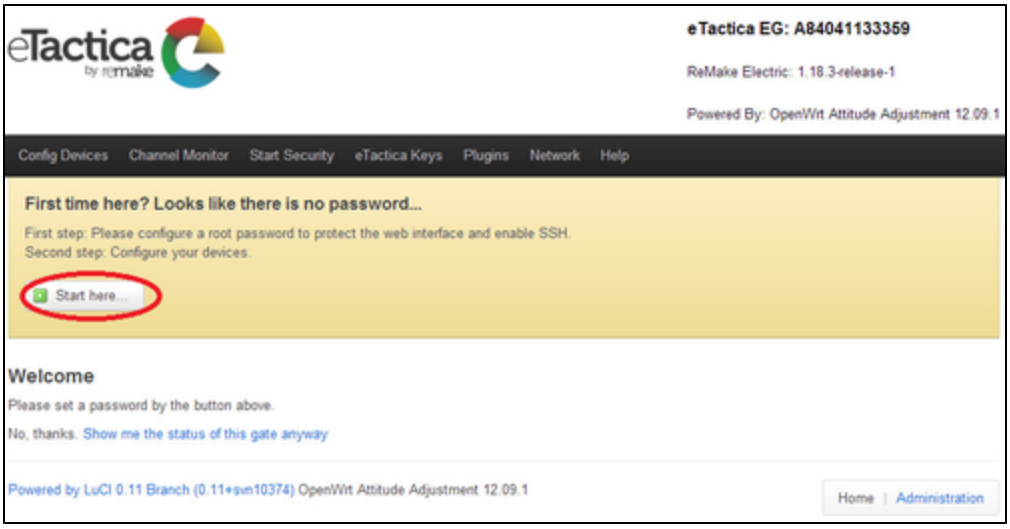
On Linux there are different tools available for this kind of discovery, i.e. *Avahi-discover*. You can use these tools to find your device and to the IP address (URL) it got assigned.

Once you have the IP address, you can enter it in your web-browser to access the admin console page of the gateway.

First Steps on the Connected Gate

Step 1 - Starting Wizard

You should see something like this:



The wizard process helps you configure the following items:

- The root password for your device
- Networking and WiFi passwords
- Configuring Modbus device list

If you want to configure these items manually, you may simply proceed as documented in the rest of this manual. However, the vast majority of installations should be able to use the wizard.

Simply click "Start here".

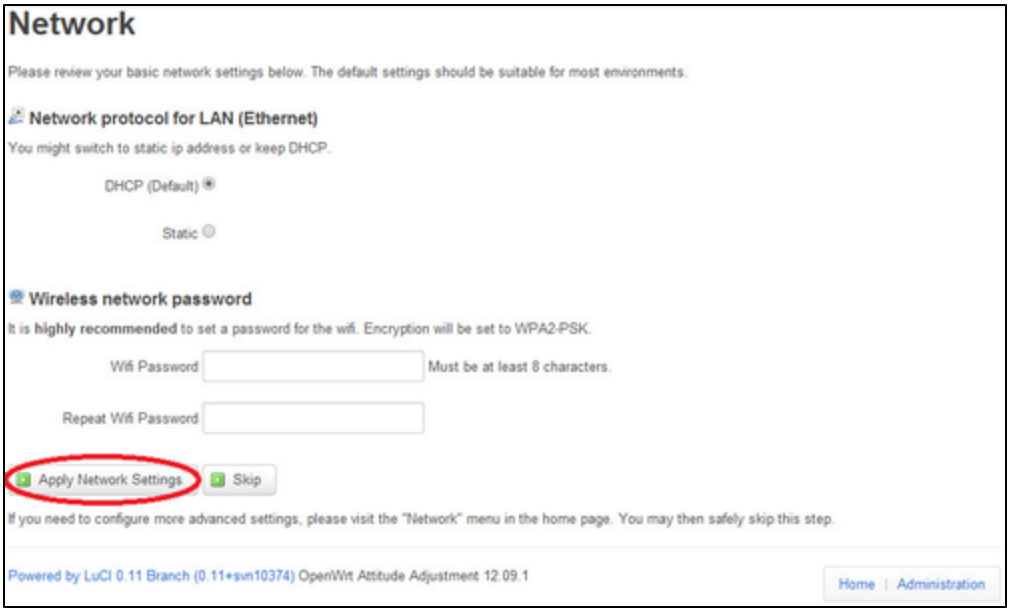
Step 2 - Setting Root Password

The root password is used to log in to the web administration console for modifying any important settings. The root password also provides SSH access to the device. You should, as always, use a good password here. Then click Configure Network for next step.

Step 3 - Configure Network

The recommended networking setup is to connect the Ethernet port to a regular DHCP network, as this requires the least configuration. Simply leave the mark on DHCP and move down to the WiFi password.

In either case, you should also enter a WiFi password here. This will use WPA2/WPA2-PSK, the best available wireless security at this time. This should be perfectly reasonable for most use cases, so choose a password and click Apply Network Settings.



Note:

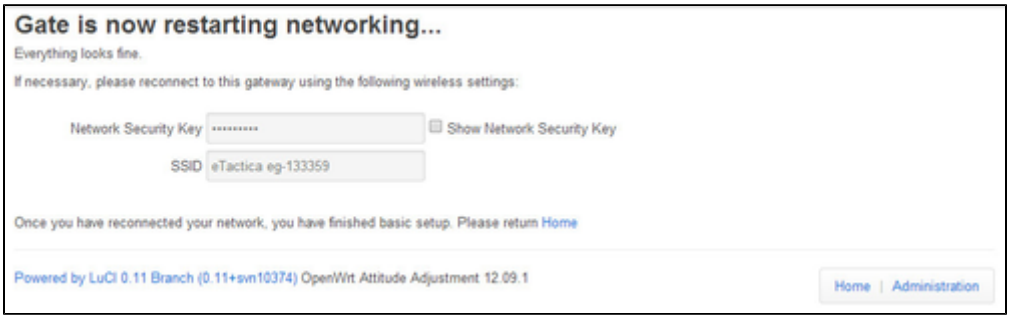
If you wish to completely disable WiFi, that is of course possible, please see page # for instructions.

Until you have reconnected with any updated networking settings, it's simply too unsafe to turn off the WiFi this early in the configuration process.

Step 4 - Reconnect

Once you have entered your desired networking setup and WiFi password, the device networking will restart.

Depending on how you had originally connected to the device, you will most likely have to reconnect. The WiFi SSID will be shown, to help you reconnect via WiFi. This may take a minute or two to restart, so please be patient.

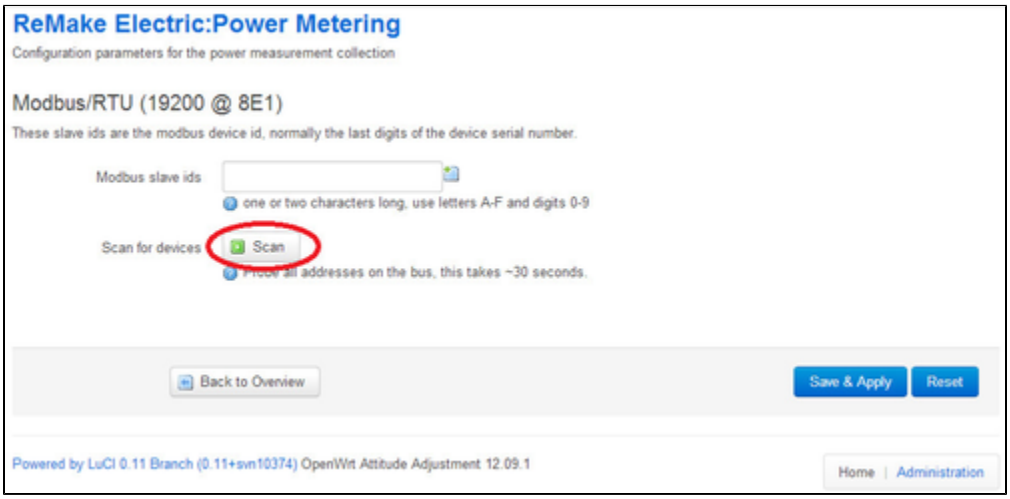


Once you have reconnected to the device, you should see a new home page.

Step 5 - Device Configuration

Now that your basic networking and security is setup, it's time to proceed to configure your measurement devices.

Select "Next: Config Devices" and you will see the following screen.



Step 6 - Scan for Devices

If you have many devices and they are all ReMake Electric devices, you can attempt to scan for all connected devices.

You should always review scan results to be sure they match the devices you expected to be found. If you choose to scan, simply press the "Scan" button.

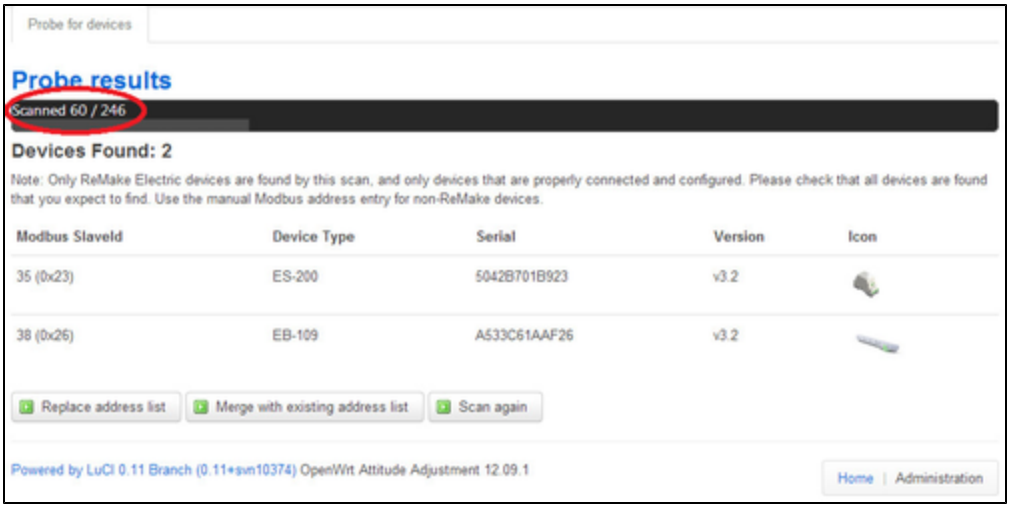
The process will take about 30 seconds, as it scans all possible Modbus addresses looking for ReMake devices.

Note:

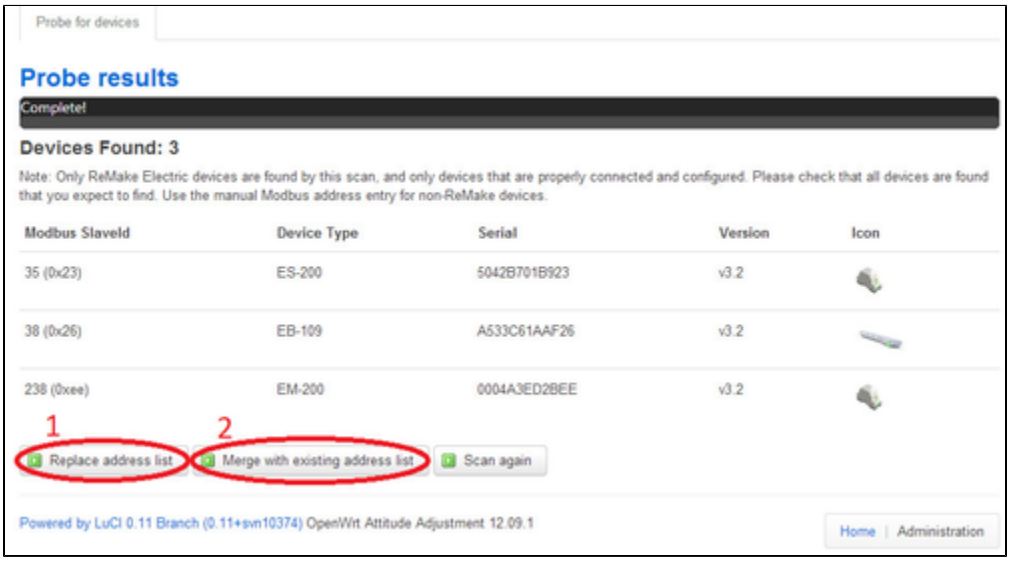
This only works for ReMake devices and only for devices that are properly connected.

Here is a screenshot of the process about half complete.

Note that it shows the Modbus address (slave ID) of the detected device, its device type, the unique serial string and an icon for each device found to help you match against what you expected.



When the scan has finished you should see all connected ReMake devices.

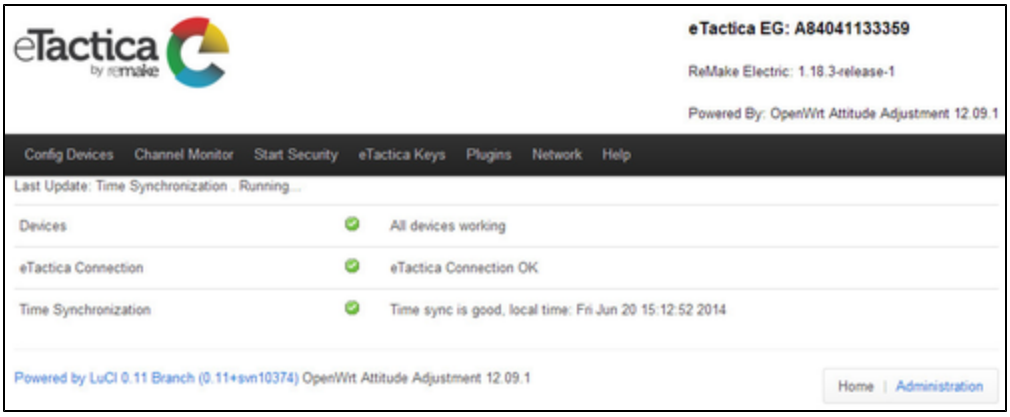


Step 7 - Saving Configuration

If you had third party devices already in your list, or if you have eTactica devices you plan on connecting later that you had manually entered in the previous step, then choose 2) "*Merge with existing address list*" to merge a combined device list. If you only care about the devices that were successfully scanned, you can choose 1) "*Replace address list*" to replace any existing list with your scan results.

If a device is not showing up in the scan list, please recheck it's wiring and power supply, and feel free to scan again.

When choosing either "*Replace address list*" or "*Merge with existing address list*", the configuration will be saved and applied. This will then take you back to the home page of the administration web console, for final diagnostics of your configuration.



Hopefully you will see three green ticks that mean that everything is working correctly:

- Devices - All devices from your configuration list are connected and recognized
- eTactica Connection - Your network settings are correct and you are successfully connected to the eTactica web application
- Time Synchronization - You have access to a NTP server that will secure correct timestamp of your measurement data

If you see red ticks on any of the above, you can go to the Troubleshooting chapter to look for a solution to your problem.

Step 8 - Completed

This completes your configuration, using the simple Wizard step by step guide.

Further Configuration

If you need to do some further configuration see the following chapters.

To edit most settings, you will need to be logged in and you will be presented with a screen like this:

The username is ALWAYS `root`, and the password is whatever you have chosen.

Configure remote MQTT bridges

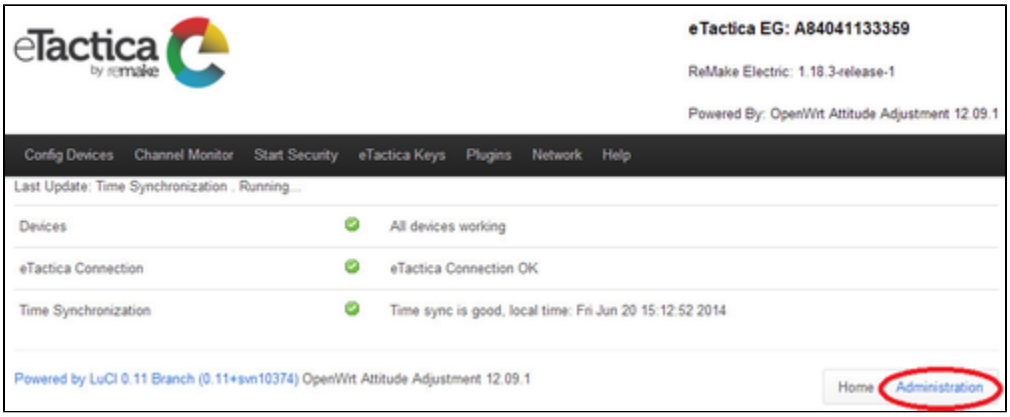
The onboard MQTT message broker, mosquitto, allows configuring multiple remote bridges to send/receive topic trees to an external broker. The UI provides some limited support for configuring these. For full details and more information you are *strongly* advised to consult the mosquitto man pages. This mechanism is how data is sent to eTactica for instance.

Step 1 - Connect to the Gateway

See Connecting to Gateway and Simple Setup

Step 2 - Go to Administration page

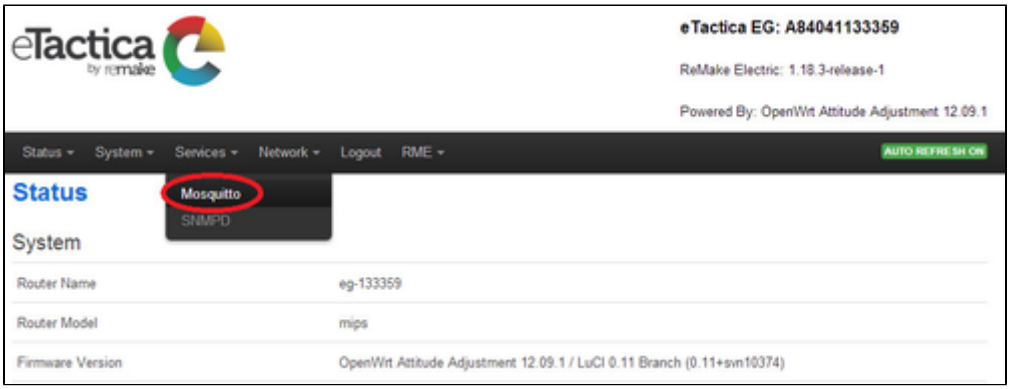
Click on "Administration"



> You will be asked to login, if you haven't already done so. See Connecting to Gateway and Simple Setup

Step 3 - Go to the mosquitto page

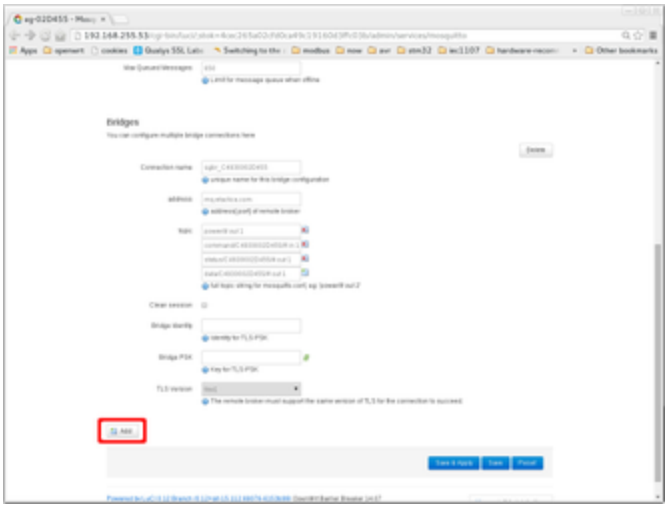
Choose **Services->Mosquitto**



Step 4 - Edit/Add/Remove bridges

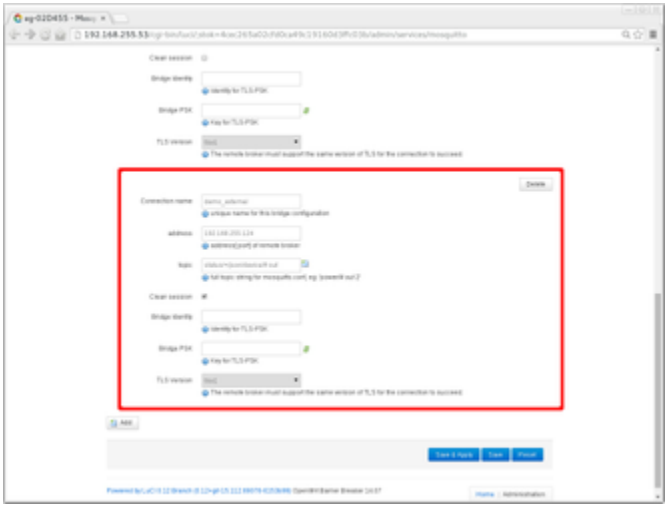
Scroll down to the section labelled "Bridges" Each bridge section can be quite large. The "delete" button will remove an entire bridge section, use caution! Click on "Add" near the bottom to create a new bridge.

> Please do *not* modify the existing bridge configuration, it is used for sending data to eTactica. It is normally recreated on every reboot if it has been modified, but to avoid confusion, we recommend simply leaving it alone.



You can add as many extra bridges here as you like (you only need one bridge per remote server, the bridge configuration can map as many topic trees as you like) The options here are a subset of those described in the mosquitto.conf man pages. Please consult that manual for advice. This configuration can be very open ended, including topic remapping and we cannot provide any concrete guidance here without more information on a given client's particular needs. If a particular configuration file option is not exposed in the UI, please file a ticket with us and we can get it added.

A simple example of sending the live readings stream to a third party broker is shown below.



Step 5 - Save settings

When you are happy with your settings, choose "Save and Apply", and the broker will restart with the new settings.

Connect to the Gateway's MQTT Message Broker as a client

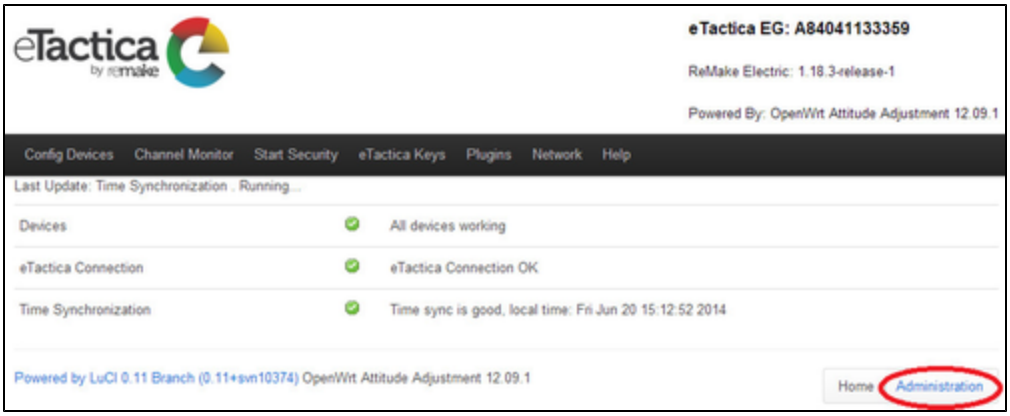
The EG runs an MQTT message broker for integration. All live readings are published to the broker as they are collected, as well as any generated alerts and configurations. If you wish to connect to the broker with your MQTT client software, you will need to at least allow external access to the broker.

Step 1 - Connect to the Gateway

See Connecting to Gateway and Simple Setup

Step 2 - Go to Administration page

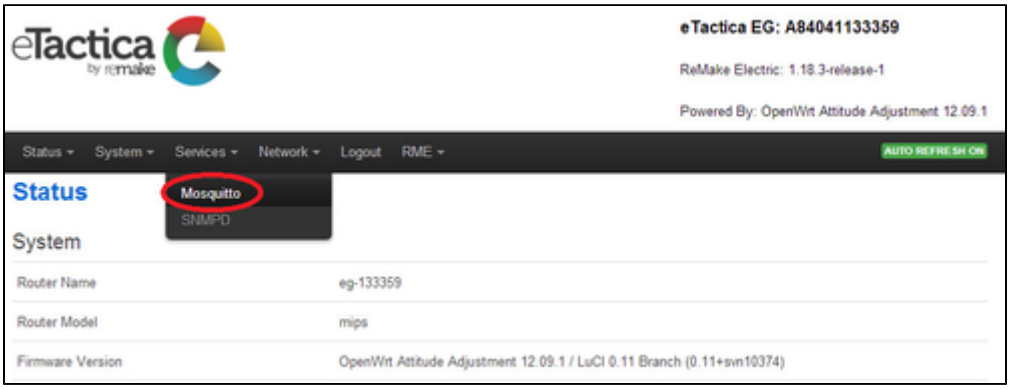
Click on "Administration"



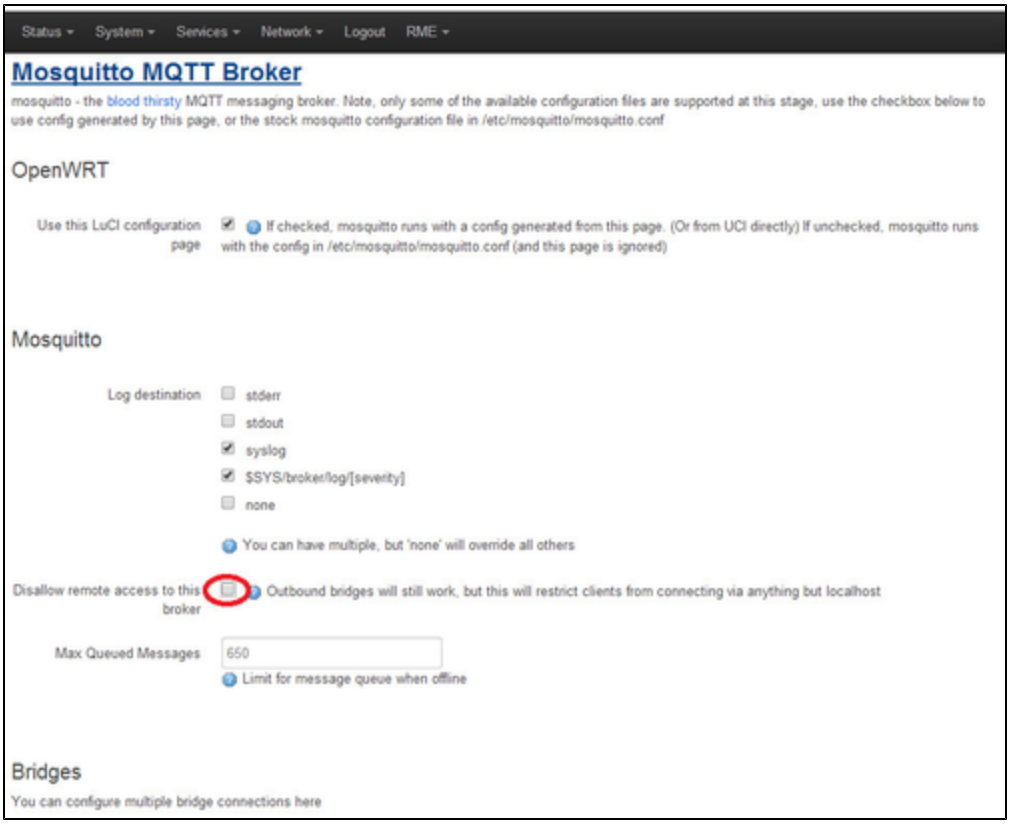
You will be asked to login, if you haven't already done so. See Connecting to Gateway and Simple Setup

Step 3 - Go to the mosquitto page

Choose **Services->Mosquitto**



Step 4 - Uncheck "Disallow remote access to this broker"



Step 5 - Save settings

Choose "Save and Apply"

> The settings here allow open access to your message broker by anyone with network access to your device. This may or may not be acceptable in your environment. You are welcome to configure the mosquitto broker with alternative security mechanisms if you prefer, but that is entirely at your discretion. Please see the mosquitto user manual for more information.

You should now be able to connect to the Gateway's MQTT broker using any MQTT client software, provided you have open network access to the device. We recommend mosquitto (see http://mosquitto.org/download/), but you can find many more options at https://github.com/mqtt/mqtt.github.io/wiki/tools

```
# Listen to all messages published on the gateway
$ mosquitto_sub -h <gateway_ip_address> -t '#'

# Listen to messages that are sent to eTactica
$ mosquitto_sub -h <gateway_ip_address> -t "power/#"

# Listen to the live data stream (Same data as uses for the Channel
Monitor page)
$ mosquitto_sub -h <gateway_ip_address> -t "status/+/json/device/#"
```

Using the SDK to create your own applications

Each release of the eTactica Gateway software includes the standard OpenWrt SDK which can be used for creating your own packages or software for installation on your EG. OpenWrt provides documentation for this process but we'll give an example here.

In many cases, you can simply create some lua scripts and use the available libraries, without having to make any packages or compile any new software. See XXXXX FIXME XXXX need to write this page with example snippets!

> This guide is written for EG200, if you are using EG100, references to "ar71xx" should be replaced with "atheros"

Requirements

- Linux build environment
- Decent understanding of compiling and software packaging

Copy the demo package feed to your build machine

Easiest is to clone the following github repository to your machine: https://github.com/remakeelectric/feed-demo

For the purpose of these steps, we're assuming you cloned the repository to /home/karlp/demos/feed-demo

Unpack the SDK and update feeds

For each release, look for the "SDK" tarball, eg http://packages.etactica.com/releases/gateway-2.8.1-release-1/targets/ar71xx/generic/openwrt-sdk-18.06-SNAPSHOT-ar71xx-generic_gcc-7.3.0_musl.Linux-x86_64.tar.xz

After unpacking this, copy feeds.conf.default to feeds.conf, and add a line for your local custom feed. Eg, for this demo add the line

```
src-link custom /home/karlp/demos/feed-demo
```

Update the feeds to include packages from this feed

```
$ ./scripts/feeds update -a
.... lots of output ....
$ ./scripts/feeds install -a -p custom
Installing all packages from feed custom.
Installing package 'hello_world'
$
```

Note that our sample "hello_world" application was "installed". Installed simply means it's available to be selected for building. You can think of it as "installed" into the SDK environment.

Build your application

By default, all "installed" applications will be built by simply running "make"

```
$ make
#
# configuration written to .config
#
 make[1] world
 make[2] package/compile
 make[3] -C /home/karlp/demos/feed-demo/hello_world compile
 make[2] package/index
$
```

If all goes well, you now have a new binary package suitable for installation on your EG in the "bin/packages/mips_24kc/custom"

Install and run your application

You can now copy your binary package to your device.

```
$ scp bin/packages/mips_24kc/custom/hellodemo_1-1_mips_24kc.ipk
root@192.168.255.74:/tmp
Warning: Permanently added '192.168.255.74' (RSA) to the list of known
hosts.
root@192.168.255.74's password:
hellodemo_1-1_mips_24kc.ipk
100% 2145     2.1KB/s   00:00
$
```

Now SSH to your device and install it, and you can test running it.

```
root@eg-037BCC:~# opkg install /tmp/hellodemo_1-1_mips_24kc.ipk
Installing hellodemo (1-1) to root...
Configuring hellodemo.
root@eg-037BCC:~# hello
Hello!
root@eg-037BCC:~# hello sdk-user
Hello demo: sdk-user
root@eg-037BCC:~#
```

Clearly, this only scratches the surface of what can be done. You could also build and install any existing OpenWrt package from any other feed as well.