DATA MODEL STRUCTURE PROPOSAL FOR OPENWRT

Increasing its' popularity among industry players, OpenWRT has the potential to be the base operating system for the majority of the devices in residential gateway market. Being one of these industry players and have been developing its' own OpenWRT based OS for over five years now, Inteno is well-informed about the great value OpenWRT brings to the table as well as the pieces it has been missing in order to be the undisputed choice as OS for the residential gateway world.

UBUS has been a great amendment to OpenWRT helping us to build applications that can communicate with each other and build on-top applications that can access and control the device in a controllable fashion (via RCPD). At Inteno, we do greatly take advantage of UBUS by making sure all our applications hook up to ubus and create objects allowing us to control the entity they are related to. As you can see below, today, in our default software, there are more than 80 objects and only 15 of them are default OpenWRT objects. This number will definitely keep growing as we develop more applications.

```
root@Inteno:/# ubus list | wc -l
86
```

UCI, configuration database of OpenWRT, on the other hand is another core utility that we greatly depend on as we make every component of our HW/SW configurable via it. As you can see below, in our default software we have more than 40 UCI configuration files and only 15 of them are from OpenWRT.

```
root@Inteno:/# ls /etc/config | wc -l
41
```

Being a devoted user of these two core components of OpenWRT, we believe there are two obstacles in OpenWRT on its' way to be the inevitable choice for the majority of the gateway vendors:

Configuration of an entity and data collection from the same entity cannot be achieved via single object: For any on-top applications Inteno or third party developers build (WebGUI, remote applications etc.) we have to interact with both UBUS and UCI (libuci or uci object on ubus) to be able to control a single entity. This makes it more difficult for developers as they have to know not only what ubus objects but also what uci files and sections they must interact with, and interacting with two components increases the number of the code they have to write. It also makes it quite complicated in terms of access control towards an entity of the software.

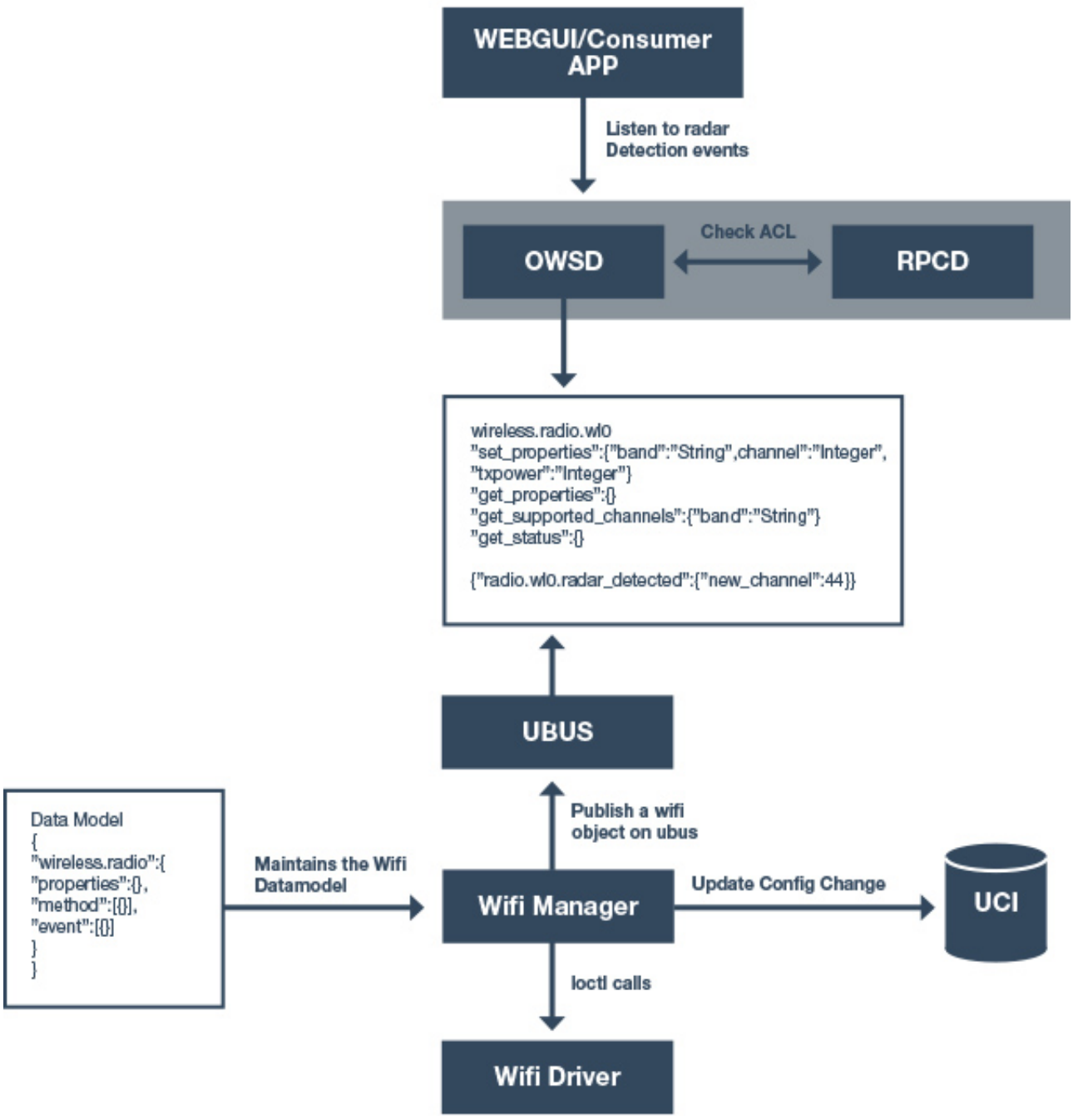
There are no ubus data models defined for controlling the entities: If there were data models defined for how to configure/control the different entities of the software (network, wireless etc.), any application built on top of this data model would seamlessly work across different platforms and OpenWRT based systems as long as the applications controlling these entities were following the data model. Moreover, a defined data model for ubus based on a data model structure which addresses the first problem, UCI would no longer be mandatory which makes OpenWRT much easier to adapt to for many vendors out there.

We believe the solution to these problems would be developing a data model structure towards OpenWRT's ubus with features such as:

- JSON based, well formatted, easy to use/understand
- supports auto code and document generation

- contains version, properties (configuration), methods, valid data types/ranges, error codes per object
- allows configuration, data collection and actions within the same object as opposed to OpenWRT today where ubus methods + UCI (using ubus uci object or libuci or /sbin/uci directly) are used.

The figure below demonstrates the flow of how a defined data model is converted to an ubus object to be used by an on-top application. (Note: OWSD is replacement to UHTTPD in Inteno's software.)



Given the example data model definition, wireless.radio, below:


```
{
	"wireless.radio": {
		"properties": {
			{
				"description": "get name of the radio",
				"name": "name",
				"type": "s",
				"access": "read"
			},
			{
				"description": "set/get band of the radio: a, b, auto",
				"name": "band",
				"type": "s",
				"access": "readwrite"
			},
			{
				"description": "set/get channel of the radio",
				"name": "channel",
				"type": "i",
				"access": "readwrite"
			},
			{
				"description": "set/get txpower of the radio",
				"name": "txpower",
				"type": "i",
				"access": "readwrite"
			}
		},
		"method": [
			{
				"description": "get the channels supported by the radio on the given specific band",
				"name": "get_supported_channels",
				"input": {
					{
						"name": "band",
						"type": "s"
					}
				},
				"output": {
					{
						"name": "channels",
						"type": "ai"
					}
				}
			},
			{
				"description": "get current radio channel, bandwidth, noise and rate information",
				"name": "get_status",
				"output": {
					{
						"name": "channel",
						"type": "i"
					},
					{
						"name": "bandwidth",
						"type": "i"
					},
					{
						"name": "noise",
						"type": "i"
```

```
                                        },
                                        {
                                                "name": "rate",
                                                "type": "i"
                                        }
                                }
                        }
                ],
                "event": [
                        {
                                "description": "event created upon radar detection by the radio",
                                "name": "radar_detected",
                                "data": {
                                        "name": "new_channel",
                                        "type": "i"
                                }
                        }
                ]
        }
}
```

and an example UCI wireless config below:

```
config wifi-device 'ra0'
        option band 'b'
        option channel '1'
        option htmode 'HT20'
        option country 'DE'

config wifi-iface
        option device 'ra0'
        option network 'lan'
        option mode 'ap'
        option ssid 'TEST-2.4G'
        option encryption 'none'

config wifi-device 'rai0'
        option band 'a'
        option channel '36'
        option htmode 'VHT80'
        option country 'DE'

config wifi-iface
        option device 'rai0'
        option network 'lan'
        option mode 'ap'
        option ssid 'TEST-5G'
        option encryption 'none'
```

an application responsible for WiFi management, will, based on the given data model structure and UCI wireless config, create the ubus objects below:

```
wireless.radio.ra0
        "set_properties":{"band":"String",channel":"Integer", "txpower":"Integer"} or "set_properties":
{"properties":"Table"}
        "get_properties":{}
        "get_supported_channels":{"band":"String"}
        "get_status":{}
```

wireless.radio.rai0
"set_properties":{"band":"String",channel":"Integer", "txpower":"Integer"} or "set_properties": {"properties":"Table"}
"get_properties":{}
"get_supported_channels":{"band":"String"}
"get_status":{}

In the first set_properties example, all properties are converted to arguments with the defined types for set_properties method.

In the second set_properties example, set_properties method accepts an JSON table as argument.

In both cases, the goal is to allow configuration of the respective radio using the wireless.radio.X object without requiring to interact with UCI directly.

Besides configuration and methods, even though, in ubus, events cannot be tied to objects, we envision that according to data model above, the event below will be available on bus upon radar detection for interested parties to subscribe to.

{ "radio.rai0.radar_detected": {"new_channel":44} }

If we take an example from an already existing OpenWrt ubus object structure: network.interface

config interface 'lan'
option type 'bridge'
option proto 'static'
option ipaddr '192.168.1.1'
option netmask '255.255.255.0'
option ip6assign '60'
option ifname 'eth1 eth2 eth3 eth4'

In current OpenWrt, the UCI network config interface section above read by netifd results in network.interface.lan ubus object as below:

root@OpenWrt:/# ubus -v list network.interface.lan
'network.interface.lan' @8f362e59
"up":{}
"down":{}
"status":{}
"prepare":{}
"dump":{}
"add_device":{"name":"String","link-ext":"Boolean"}
"remove_device":{"name":"String","link-ext":"Boolean"}
"notify_proto":{}
"remove":{}
"set_data":{}

With the data model structure we suggested above, it would look like:

'network.interface.lan
"set_properties":{"properties":"Table"} or "set_properties":{"type":"String", "proto":"String", "ipaddr":"String", "ip6assign":Integer, ...…}
"get_properties":{}
"up":{}
"down":{}
"status":{}
"prepare":{}
"dump":{}
"add_device":{"name":"String","link-ext":"Boolean"}
"remove_device":{"name":"String","link-ext":"Boolean"}
"notify_proto":{}
"remove":{}
"set_data":{}

Where ubus call uci get '{"config":"network","section":"lan"}' returns below:

```
{
	"values": {
		".anonymous": false,
		".type": "interface",
		".name": "lan",
		"type": "bridge",
		"proto": "static",
		"ipaddr": "192.168.1.1",
		"netmask": "255.255.255.0",
		"ip6assign": "64",
		"up": "1",
		"device": "eth1 eth2 eth3 eth4",
		"ifname": "br-lan"
	}
}
```

ubus call network.interface.lan get_properties could, for example, return:

```
{
	"properties": {
		"type": "bridge",
		"proto": "static",
		"ipaddr": "192.168.1.1",
		"netmask": "255.255.255.0",
		"ip6assign": "64",
		"up": "1",
		"device": "eth1 eth2 eth3 eth4",
		"ifname": "br-lan"
	}
}
```

In the reverse direction, passing the properties (in form of separate parameters or table) above as argument to set_properties method of network.interface.lan object would re-configure the LAN interface.

Even though netifd is talking to UCI in order to set/get properties, from the perspective of applications built on top, they require access to network.interface.lan object only in order to control LAN interface. If a vendor has their own network implementation and configuration mechanism, but still wants to adapt OpenWrt, they just have to make sure they make the network.interface.X objects available on UBUS.

Supporting such a data model structure, the ultimate goal would be to create an OpenWrt data model to be standardized (ubus objects for controlling network, wireless, voice etc.) in order to achieve compatibility across different OpenWrt based systems. With this achieved, device vendors or others who create OpenWrt based firmware would have the freedom to use their own applications and configuration mechanisms provided that their OpenWrt implementation creates the required ubus objects following the defined data model. Not only any application built on top of this defined data model will seamlessly work across different systems but also the well formatted and documented data model structure will allow third party and remote app developers independently build their applications thus reduce the time needed to support them.

The objects created according to the defined data model could have a prefix in order to differentiate them from the custom objects.

We believe, not only Inteno but almost the whole OpenWrt community could benefit from having a well formatted and documented data model structure to be used in creating ubus objects as well as code/document generation. Making configuration of a specific entity available via its ubus object would definitely facilitate building on-top applications.