May 2021

FAAS/SERVERLESS-BASED LIGHTWEIGHT SOFTWARE DEVELOPMENT KIT FRAMEWORK FOR 6LOWPAN ARCHITECTURES

Lele Zhang

Chuanwei Li

Li Zhao

Harbor Dong

FAAS/SERVERLESS-BASED LIGHTWEIGHT SOFTWARE DEVELOPMENT KIT FRAMEWORK FOR 6LOWPAN ARCHITECTURES

AUTHORS:
Lele Zhang
Chuanwei Li
Li Zhao
Harbor Dong

ABSTRACT

Effecting application development and/or updates in a wireless mesh networking environment raises a number of challenges. To address such challenges techniques are presented herein that support an extensible serverless framework that provides a function as a service (FaaS) paradigm for the rapid development and the easy operation and maintenance for Internet of things (IoT) customers. Among other things, a customer's application code may be uncoupled from a vendor's kernel and software development kit (SDK) library (thus freeing a customer from having to consider platform dependencies and allowing them to focus just on their business logic) and then generated as a small, platform-independent script that may be quickly and easily delivered to a massive number of endpoints. Additionally, a customer may add their own application programming interfaces (APIs) into a provided command-line interface (CLI) Commands library. As well, a customer may leverage a virtualized or remote development and simulation environment to speed up development activities.

DETAILED DESCRIPTION

Wireless mesh networking products (such as, for example, a connected grid mesh network (CG-Mesh), a Wireless Smart Utility Network (Wi-SUN), etc.) have existed for many years for industrial applications, such as electric/water/gas meters, street lights, intelligent farming, secure mine management, etc. Such networks consist of a massive number of resource-limited wireless sensors and are typically formed as a tree-like topology based on, for example, the Institute of Electrical and Electronics Engineers (IEEE)

802.15.4 technical standard and the Routing Protocol for Low-Power and Lossy Networks (RPL, as defined in the Internet Engineering Task Force (IETF) Request for Comments (RFC) 6550). In general, a customer develops their applications based on a CG-Mesh or Wi-SUN framework by using a vendor's SDK or APIs. Figure 1, below, presents aspects of the detailed architecture of such an approach.

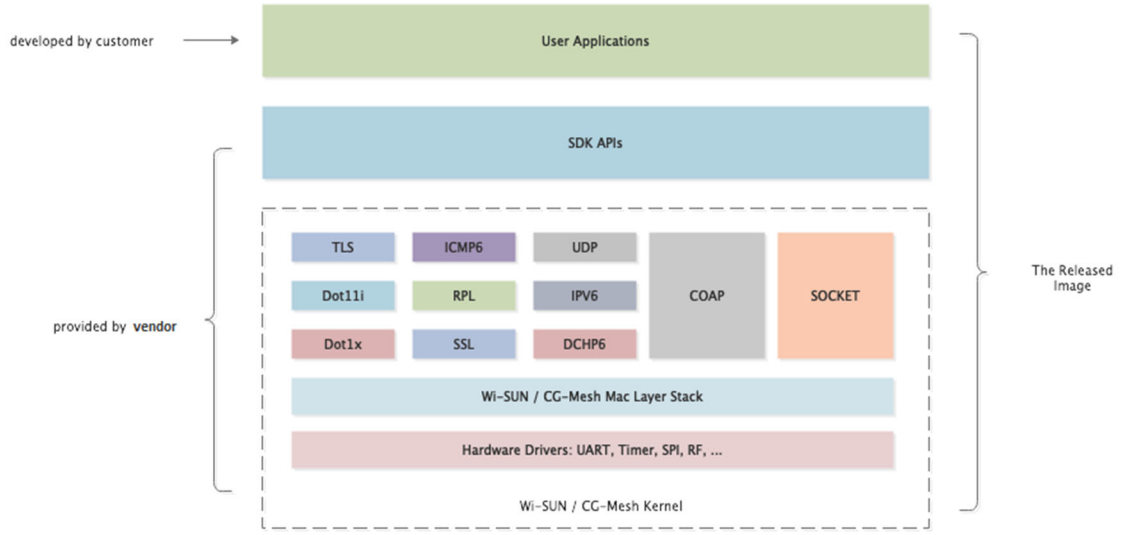

Figure 1: Illustrative Detailed Architecture

A customer typically combines their application code with a vendor's kernel and SDK library in order to generate a final image, the size of which is often too large. Frequently, the main body of a target image is comprised principally of a vendor's kernel and SDK library (e.g., almost 90% of the final image) with the kernel and the SDK typically unchanged. Therefore, a customer is still required to generate a large sized image even if they just make a very small change to their application. Thus, the cost of a firmware upgrade for a large-scale Low Power and Lossy Network (LLN) deployment may be excessive (e.g., it may involve several days to upgrade the firmware for a personal area network (PAN) which may comprise almost 5,000 nodes).

Figure 2, below, illustrates why such an upgrade process may take so long. As shown, the firmware upgrading may only be propagated in a broadcast (BCAST) slot. But, the BCAST schedules are not used just for firmware upgrading, they are also needed to propagate various network maintenance artifacts such as, for example, destination-oriented

directed acyclic graph (DODAG) information object (DIO) messages. Therefore, thousands of image blocks are broadcast with a low priority. Additionally, each message needs to be transmitted multiple times to ensure reachability because there is no acknowledgement for a BCAST message. Consequently, the customer suffers from the inconvenience of regular firmware upgrading making operation and maintenance difficult.

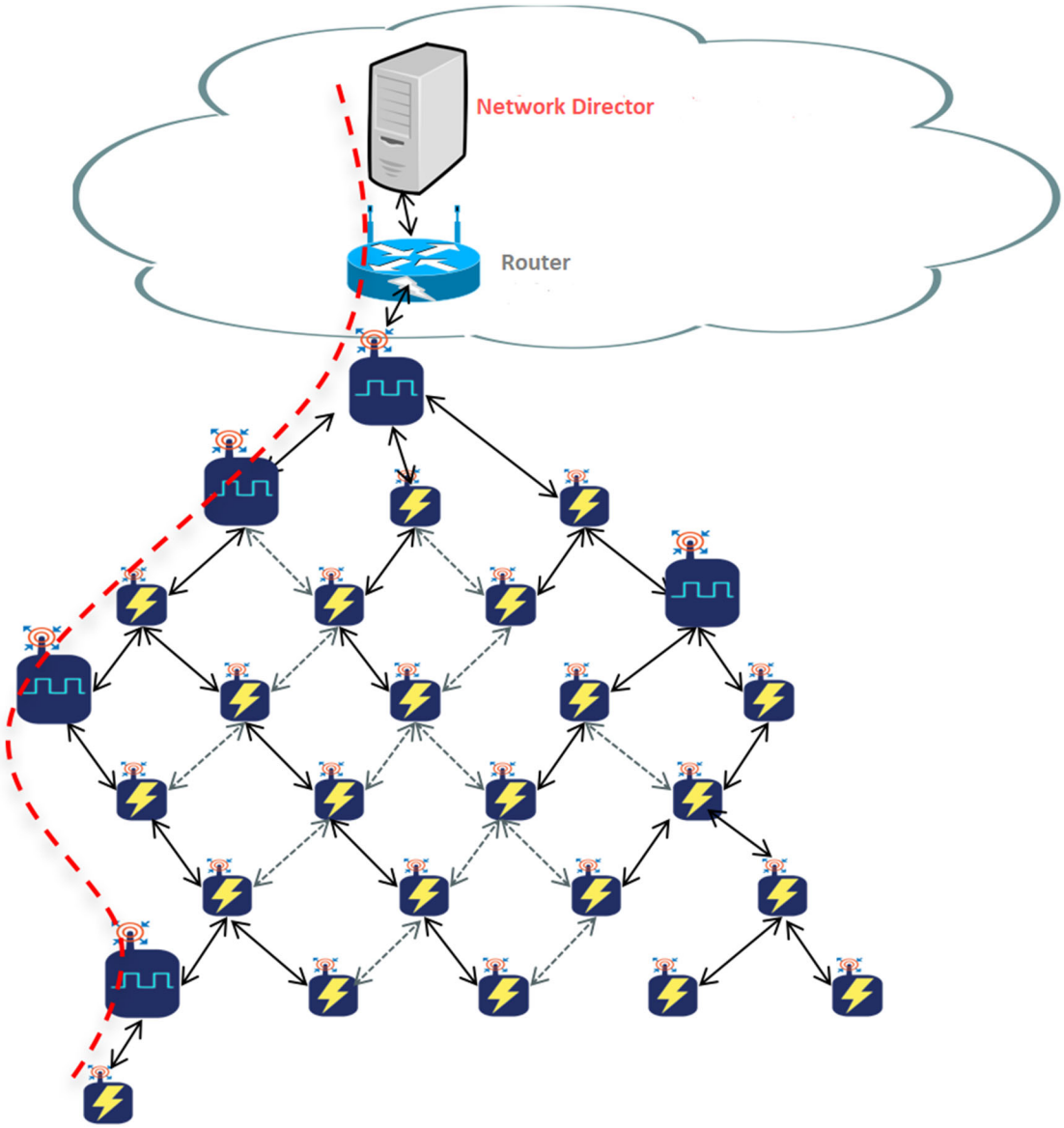

Figure 2: Illustrative Firmware Upgrade Process

Another challenge concerns diversified development tools and an exponentially increasing number of software versions. A customer often has several hardware products for different business purposes, each of which may use a different microcontroller (MCU) or other peripheral chips (such as, for example, a radio receiver). Consequently, corresponding integrated development environments (IDEs) or compilers are needed. Thus, even though the application code may be the same it is still necessary to complete a build process multiple times, for each of the different platforms. Alternatively, the same platform may have different bodies of application code leading to multiple versions needing to be maintained. Additionally, customer developers may need to know further technologies regarding embedded devices even though their applications are not related to those area. Figure 3, below, depicts aspects of the software version maintenance challenges that were just described.

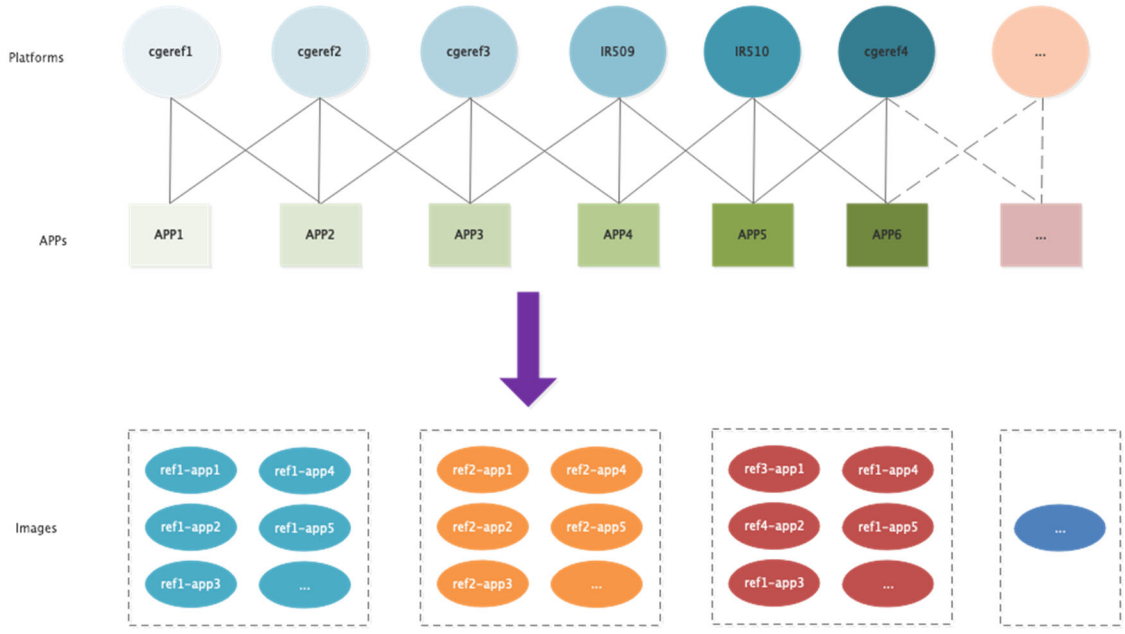

Figure 3: Software Version Maintenance Challenges

To address the types of challenges that were described above, techniques are presented herein that support a serverless framework that provides a FaaS paradigm for rapid development and easy operation and maintenance for Internet of things (IoT) customers. Aspects of the techniques presented herein enable, among other things:

1. The uncoupling of user application code from a vendor's kernel and SDK library, thus reducing firmware upgrade cost since a kernel and SDK do not also need to be upgraded.
2. The uncoupling of user application code from a vendor's kernel and SDK library so that a customer need focus just on their business application logic block rather than on other embedded system development tasks.

Various existing approaches to the types of challenges that were described above address some, but not all, of the challenges and potentially carry a range of deficiencies.

A first existing approach proposes a framework for the management of IoT devices based on a serverless computing paradigm where all of the nodes are based on microprocessor solutions (which, importantly, are not resource constrained as are wireless sensor network (WSN) devices). Thus, this approach is not suitable for Wi-SUN products.

The approach's architecture proposes an API gateway to receive and analyze the requirement of user applications. The gateway then takes charge of executing those applications. A customer need only focus on their own business applications rather than any additional operation and maintenance management.

Aspects of the techniques presented herein apply elements of such an approach to Wi-SUN products and simplify the application development process for a customer. For example, a customer need not learn any IDE, language, or other hardware-dependent knowledge. Furthermore, a customer can even develop using a virtualized device which may speed up their development time through the use of simulations.

Figure 4, below, illustrates elements of a current development solution and elements of an enhanced development solution that may be possible under aspects of the techniques presented herein.

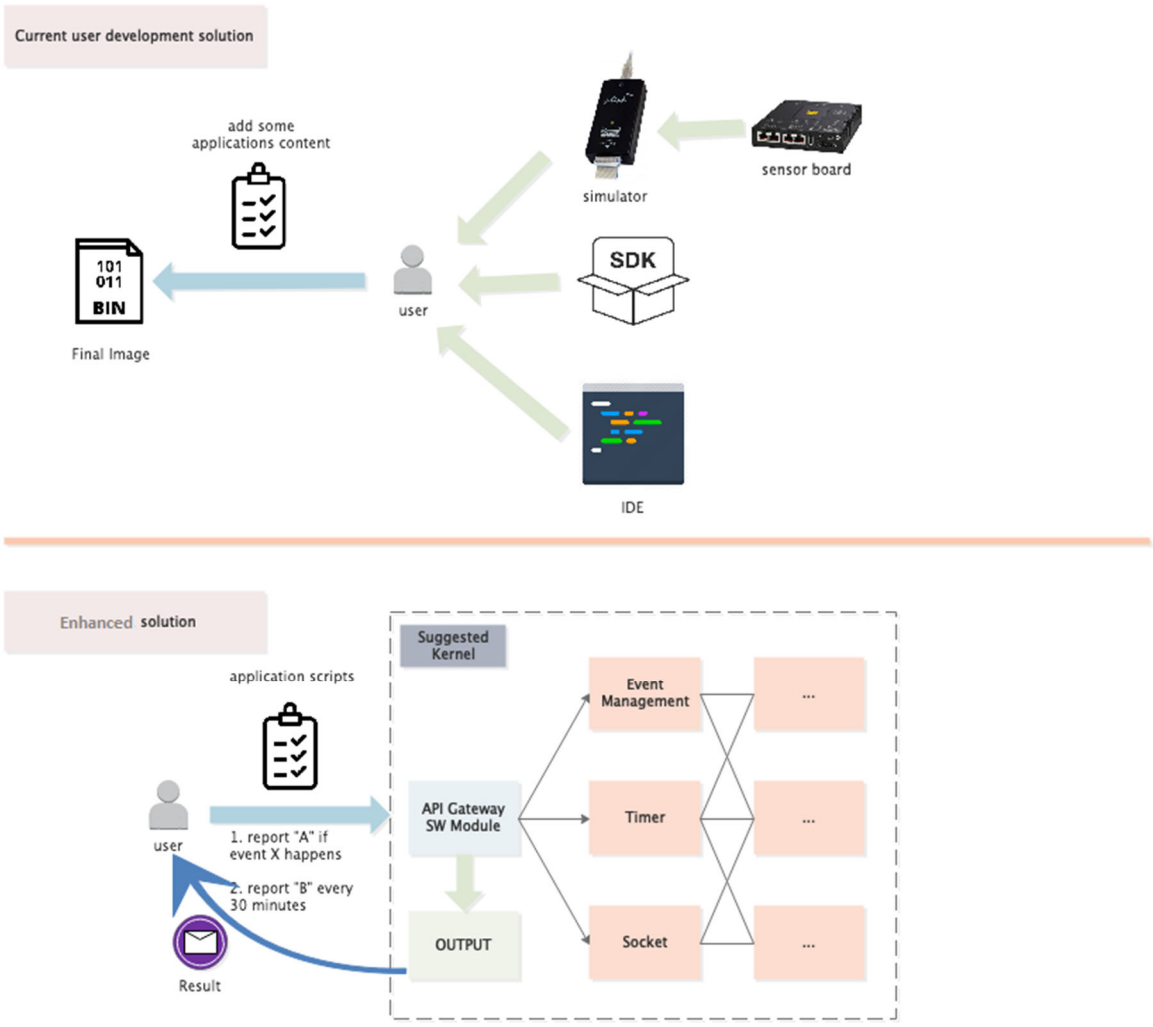

Figure 4: Enhanced Wi-SUN Product Kernel Architecture

A second existing approach encompasses the use of shell scripts. A shell script is a computer program that is designed to be run by, for example, a Unix shell (e.g., a command-line interpreter). There is a special grammar interpreter to translate a shell script into an executive programming block for the embedded terminal device. Aspects of the techniques presented herein support a serverless framework based on shell-like programming for low power and limited-resource wireless sensors.

A third existing approach employs the Constrained Application Protocol (CoAP) Simple Management Protocol (CSMP), a representational state transfer (REST) compliant command set for managing large scale deployed endpoints.

Typically, a cloud-based network director can be utilized to manage all of the endpoints by using CSMP Type Length Value (TLV) artifacts. A customer may easily configure the endpoints for performing simple tasks with TLVs (e.g., a node reports regular data to the network director every six hours but this report period will be shortened to one hour if a key parameter exceeds a certain threshold).

While CSMP could be used to modify the threshold or the report interval for remote endpoints, such an approach still has various limitations. For example:

1. There may be unexpected latency between a network director and a remote endpoint, so sometimes a remote endpoint may receive the configurations too late.
2. The approach does not work when the application workflows are totally changed as the parameter configurations cannot cover everything for various changeable user requirements.

A fourth existing approach employs the MgmtUdpCommand component in Wi-SUN products. This is an existing component which provides a simple CLI facility for speeding up internal test and troubleshooting. However, this functionality is disabled in the official release to a customer. Aspects of the techniques presented herein enable and enhance this functionality and, thus, simplify an SDK customers' development and maintenance.

A fifth existing approach supports generating a difference (i.e., diff) image for firmware upgrades in LLNs, which may reduce traffic and energy consumption in a large-scale upgrade case. While such diff images are smaller than before, they are still large (e.g., usually larger than 300 KB). Additionally, such an approach does not address the other challenges that were described above.

Aspects of the techniques presented herein address the various challenges that were described above by, among other things, supporting a special scripting language for Wi-SUN nodes that understands how to execute user applications. Employing such techniques, a customer can edit scripts rather than coding with SDKs or APIs to generate their applications. The Wi-SUN kernel has a specially designed syntax parser for terminal sensors to know how to execute user applications. Such a script component may support

a number of functionalities, including several simple but essential syntax components, including for example:

a. A ‘loop statement’ such as, for example, ‘for’ and ‘while’ in the C programming language.
b. A ‘conditional statement’ such as, for example, ‘if … else …’ in the C programming language.
c. Subroutines and ‘main function entry.’

The script component is not intended to support multiple tasks or multiple threads due to limited resources and very simple user requirements (e.g., an automatic meter reading). A user may employ a special function to invoke the complex functionalities that are provided by CLI command sets (e.g., MgmtUdpCommand components). For example, Figure 5, below, presents a code fragment for rebooting a device.

```
func1() {
  if (condition 1) {
    system_cmd("reboot"); // reboot the node
  }
}
```

Figure 5: Exemplary Code Fragment

It is important to note that through aspects of the techniques presented herein a user application may be decoupled from a Wi-SUN kernel and SDK and may be generated as an independent and small script that may be easily delivered to a massive number of endpoints.

Rather than generating various versions based on different hardware platforms, a single all-in-one script may be used as long as a customer’s business application requirements are the same. As depicted in Figure 6, below, a user uploads a common application script to a network director and then the network director pushes the updates to a router for distribution. Although there are multiple different hardware platforms in a PAN, they can each use a uniform script without any platform dependency.

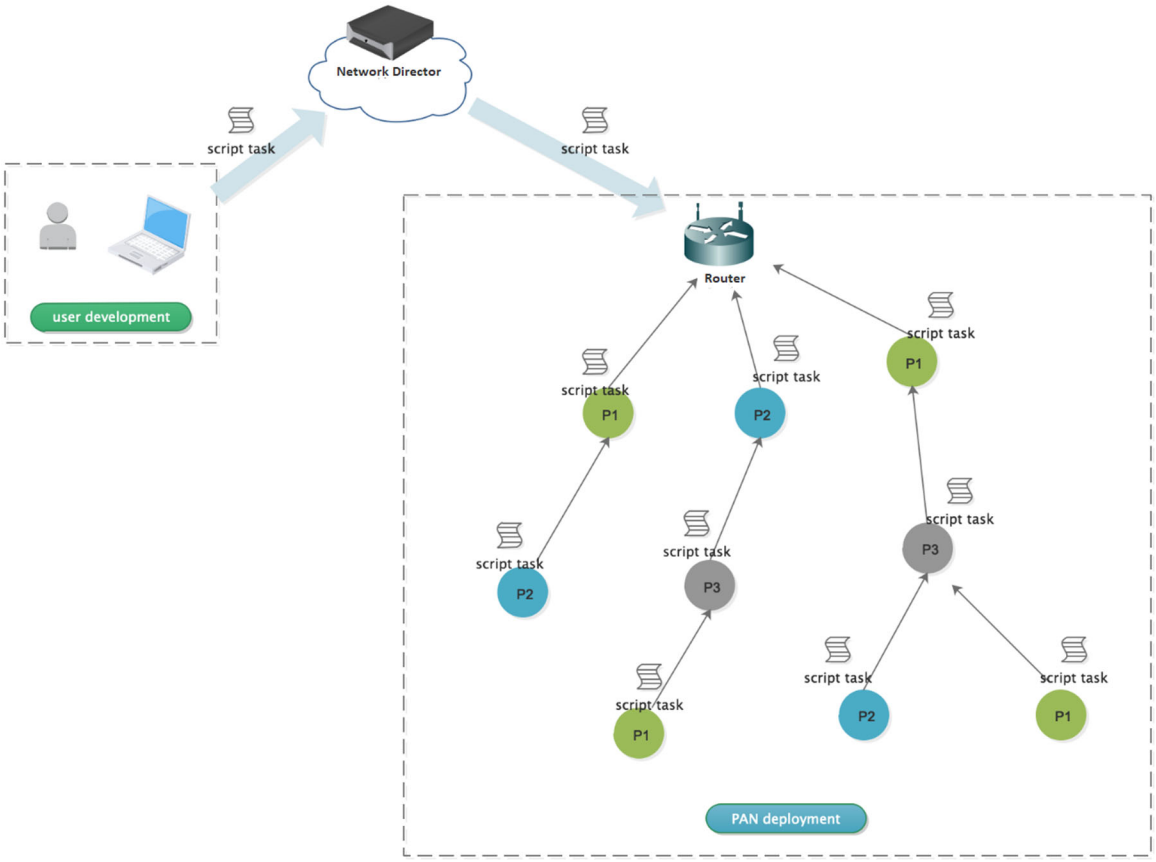

Figure 6: Exemplary Script Development and Deployment

Since the size of such scripts are usually very small their deployment consumes less traffic and reduces the time to upgrade for a massive number of nodes in a large-scale deployment.

It is important to note that through aspects of the techniques presented herein the kernel is in charge of implementing the complex modules and providing CLI Commands as SDKs and APIs as described above. A number of functionalities may be used including, but not limited to:

1. Setting and obtaining key networking parameters such as, for example, PAN identifier (PAN ID), Service Set Identifier (SSID), modulation (e.g., 2 frequency-shift keying (2FSK), orthogonal frequency division multiplexing (OFDM), etc.), band identifier, data rate, forward error correction (FEC) enable/disable, security level, device role (e.g., leaf node or coordinator), etc.

2. Operating available hardware modules such as, for example, general-purpose input/output (GPIO) elements, a timer, an interrupt, a light-emitting diode (LED), a universal asynchronous receiver-transmitter (UART), flash, a watchdog, an inter-integrated-circuit (I2C), an analog-to-digital converter (ADC), a serial peripheral interface (SPI), etc.
3. Using the Internet Protocol version 6 (IPv6) over Low -Power Wireless Personal Area Networks (6LoWPAN) protocol suite APIs for connecting with a remote server or other IoT devices (such as, for example, Internet Control Message Protocol version 6 (ICMPv6), Dynamic Host Configuration Protocol (DHCP), User Datagram Protocol (UDP), socket, Constrained Application Protocol (CoAP), etc.)
4. Using other system service commands such as, for example, reboot, firmware upgrade, ifconfig, global time, etc.

It is important to note that aspects of the techniques presented herein are extensible. For example, a customer may add their own APIs into the provided CLI Commands library as long as they follow defined development rules and guidelines. Since it is impossible to cover all situations for different industrial customers (e.g., the requirements of electricity customers may be different from those of smart lighting companies) a customer may add their application commands into the library as needed provided that such extended user-defined APIs follow defined formation rules.

For example, as depicted in Figure 7, below, if a customer wishes to add a mathematical algorithm to compute a result with sampled data from sensors, they can implement the algorithm using the C programming language and then attach it to the library by using pre-determined hook functions.

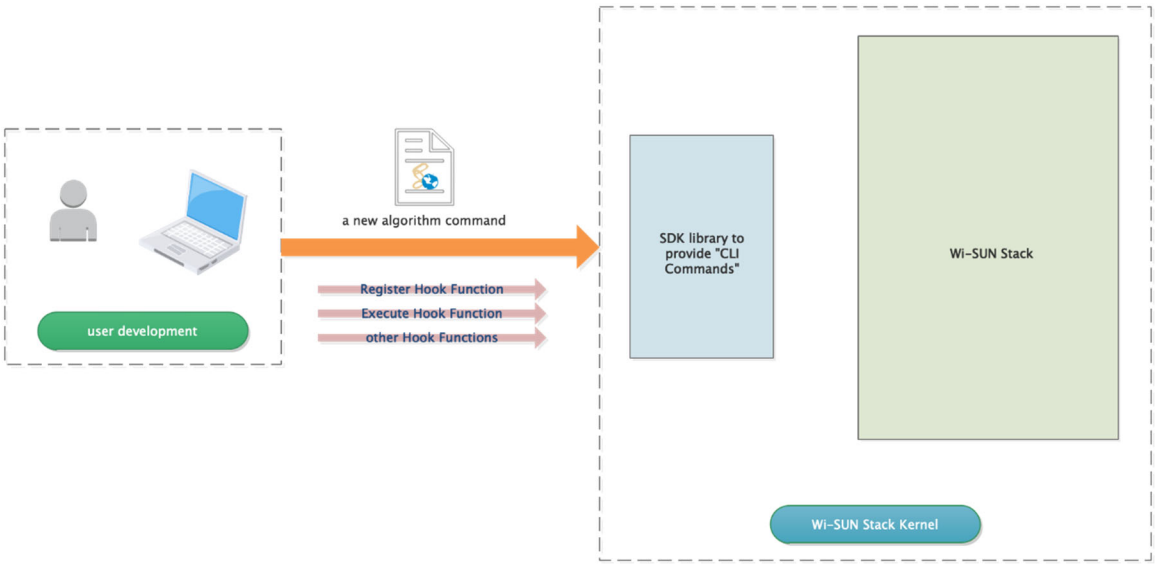

Figure 7: User-Defined API Registration

Under aspects of the techniques presented herein a cloud service may provide a virtualized or remote development and simulation environment that a customer may leverage during their development activities.

A customer application may be developed with abstract modules which do not have any hardware dependency requirements. Thus, a customer may develop their applications without real hardware (e.g., boards, devices, etc.). In a typical embedded project development life cycle, it is difficult for the software team to begin working before the real hardware devices are available. But, employing aspects of the techniques presented herein a software engineer just needs to apply for a virtualized device from a network director cloud to develop applications without any hardware dependency.

As with a traditional cloud service, a customer just needs to log in to a network director cloud service and then apply, as wanted, for one or more instances of a terminal device. For example, they could develop and then test their applications on a sandbox environment that is offered by a vendor provided such an environment has already supported such functionality. Then, the software engineer just needs to develop applications and verify them online. Virtualized or remote devices may be generated by different methods, including:

1. A pure software simulation environment. For example, a Linux-based mesh simulation product for field test, which may be referred to as The Matrix. A

customer may use this virtualized device to develop and verify their applications.

2. Actual deployed devices in a vendor's laboratory. For example, a hardware team can design and implement a test environment in the laboratory, which may be called Steel Yard and Copper Yard. It can consist of thousands of real endpoints and several real routers. A customer may employ some of the devices for developing and verifying their applications by purchasing service on the network director cloud.

Figure 8, below, illustrates aspects of the above narrative.

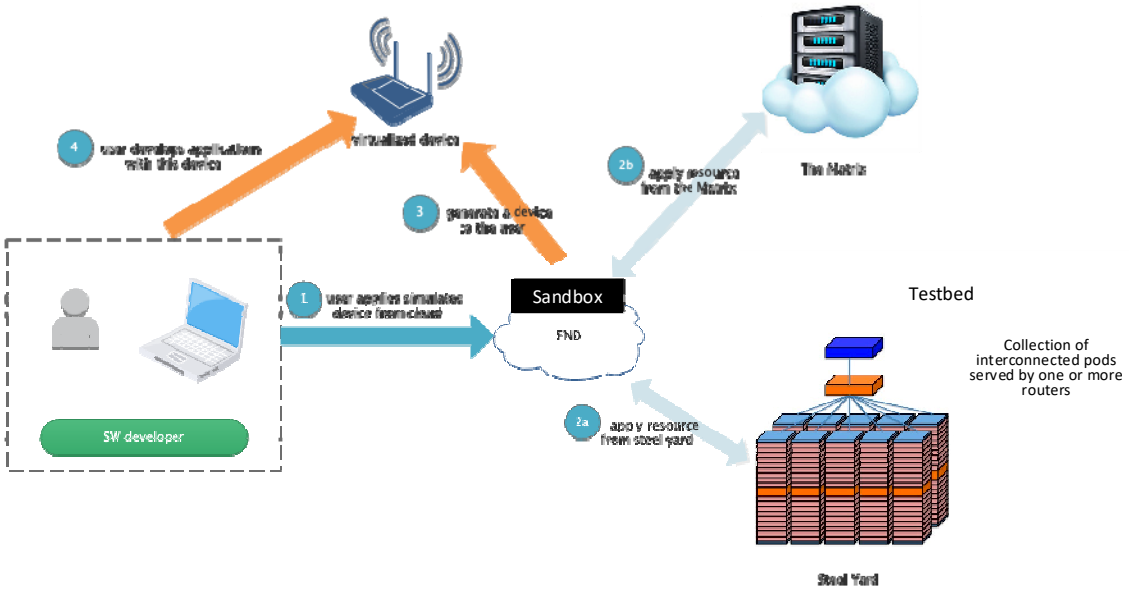

Figure 8: Illustrative Virtualized Device Resource Process

Once a customers' real devices are ready to use, they can import the verified application scripts into their deployment to test and, if needed, debug. Such an approach can shorten the development time period for customers.

To aid in the further exposition of the descriptions and the figures that were presented in the above narrative, consider an illustrative example involving a Wi-SUN environment.

Wi-SUN nodes are widely used in large-scale industrial IoT businesses, such as remote meters, smart lighting, etc. Such environments typically have a massive number of

nodes (e.g., up to thousands of nodes under a border router) in LLNs, the data rate (often 50 kilobits per second (kbps) to 150 kbps without FEC or 25 kbps to 75 kbps with FEC) is low, and the connectivity is often unstable. In addition, Wi-SUN employs two kinds of time slots to propagate datagrams – one is for unicast and the other is for broadcast or multicast – where the ratio is often three unicast time slots to one broadcast or multicast time slot. An endpoint can only use a broadcast time slot to upgrade its firmware. In order to ensure successful firmware upgrading for all nodes, each message will be propagated at least three times.

Consider an estimate for the time of a firmware upgrade for an endpoint. Assume the image size is S, the ratio of unicast time slots to broadcast or multicast time slots is r, the period of a slot is t, each block size of the image is b, the utility ratio of broadcast time slots is p, and the repeat rounds of each block is m. The formula that is presented in Equation 1, below, illustrates the time 'T' of a firmware upgrade.

$$T = \frac{\frac{S}{b} \times (r+1) \times t \times m}{p} \qquad \text{Equation 1}$$

For purposes of illustration, consider a hypothetical device whose image size is usually more than 600 kilobytes (KBs). A network director often uses 768 bytes as a block for firmware upgrading. The ratio of unicast time slots to broadcast or multicast time slots is 1:3, so r is three. A broadcast time slot period is 125 millisecond (ms) and usually one only employs 10% of broadcast time slots for firmware upgrading (the others are used for networking management – e.g., DIO datagram, MPL packets, etc.). Note that half of these broadcast time slots are used for receiving and the other half are used for forwarding on to a next hop node. Finally, each block will be repeatedly broadcast three times. Using the formula that was presented in Equation 1, above, the time for downloading a new image to the first hop is ((600*1024/768)*4*0.125*3)/0.05 seconds or about 7 hours.

Wi-SUN supports up to 24 hops and obviously there may be some amount of latency between each hop. Thus, deeper nodes need more time to receive all of the image blocks. In addition, due to poor link quality some nodes (usually at the edge of the network) cannot receive all of blocks within 3 rounds of broadcast. Such nodes may need to obtain

the missing blocks from the network director by a unicast method. Generally speaking, the failure rate is also high when unicast communication is used because of poor link quality. Additionally, unicast communication requests an acknowledgement frame from a source node, thus decreasing throughput and increasing time consumed. For example, to upgrade around 3,000 nodes in a PAN environment it may take approximately 24 hours.

Another problem is energy consumption. Because there are so many broadcast and unicast blocks or datagrams in a firmware upgrade, the nodes consume much more energy transmitting packets. Wi-SUN is a LLN where many nodes employ two AAA batteries which are expected to last a long time. Consequently, a customer may need to reduce the frequency of application updates to as low as possible even though some new features are beneficial to their business. Considering the cost of a firmware upgrade, they have to postpone the updates.

A customer's application may have a very small size but it needs to be combined with a vendor's SDK library, resulting in an image size that is large. That is dependent upon the embedded system architecture. Wi-SUN uses Cortex-M3 or -M4 chips or single chips, which require the use of an all-in-one image. Thus, the image size is always large (given the requirement of combining with a vendor's SDK library) even if a customer is just modifying a minor bug.

Through aspects of the techniques presented herein key functions from an SDK library may be summarized and a script-like interface may be provided for a customer's use.

Unlike a customer's application, the Wi-SUN SDK library is typically stable and unchanged. So, a customer just needs to develop their application scripts for their business purpose. Because the size of such script-like files is far smaller than the original image, the time consumed and the energy expended are also smaller. For example, it may be possible to upgrade 3,000 nodes in one hour. As a result, a customer can quickly develop and deploy new applications without considering the cost of a firmware upgrade.

Another significant business advantage is that a customer will often have multiple types of devices in the same places. For example, there may be advanced metering infrastructure (AMI) devices, Distribution Automation (DA) gateways, range extenders and fog devices in the same PAN, all of which support Wi-SUN standards but each of

which are based on different hardware boards. For a customer, their application is the same but they still need to test it with all of employed platforms, thus increasing their efforts. In addition, different devices must use different images (even though the applications are the same) but they must still forward received messages to their neighbors because perhaps their neighbors are using such an image. Figure 6, above, illustrated elements of this.

Through aspects of the techniques presented herein, different platforms can use the same script, resulting in, for example:

1. A significant reduction in the time expended and the energy consumed when upgrading the firmware in a large-scale deployment.
2. A focus on application development without considering the difference among multiple hardware platforms. Additionally, the testing efforts may be reduced. A virtualized test bed may even be utilized from, for example, a vendor's cloud to remotely test applications before their nodes are deployed.
3. An ability to easily manage and deploy rapid iterative applications in extra-large scaled wireless mesh networks, thus improving experience and speeding up application development.
4. A low-code implementation that can help a developer to quickly develop applications.

As noted previously, aspects of the techniques presented herein can reduce the time expended and the energy consumed during a firmware upgrade for thousands of nodes in a PAN. As a result, a customer can quickly and iteratively develop their own applications without considering the cost implications of a large-scale firmware upgrade. A customer can generate a general script-like file across different hardware platforms and a vendor can be in charge of executing their application by parsing the file. A customer can just focus on the application layer rather than caring about a hardware or platform layer, resulting in easier and/or lower cost of operation and/or maintenance in a large-scale deployment.

In summary, techniques have been presented that support an extensible serverless framework that provides a FaaS paradigm for the rapid development and the easy operation and maintenance for IoT customers. Among other things, a customer's application code may be uncoupled from a vendor's kernel and SDK library (thus freeing a customer from having to consider platform dependencies and allowing them to focus just on their business

logic) and then generated as a small, platform-independent script that may be quickly and easily delivered to a massive number of endpoints. Additionally, a customer may add their own custom APIs into a CLI Commands library. Further, a virtualized or remote development and simulation environment to speed-up application development activities.