J Reliable Intell Environ (2016) 2:131–144
DOI 10.1007/s40860-016-0027-5

ORIGINAL ARTICLE

# One IoT: an IoT protocol and framework for OEMs to make IoT-enabled devices forward compatible

**Gourinath Banda**[1] · **Chaitanya Krishna Bommakanti**[1] · **Harsh Mohan**[1]

Received: 31 May 2016 / Accepted: 2 August 2016 / Published online: 30 August 2016

**Abstract** Internet of Things (IoT) paradigm is going to imbue and become ubiquitous in everyday living, ranging from generic household, healthcare, public utility to defence applications. The IoT as a technology realm is witnessing advancement at a lightning speed. Consequently, there is a growing number of IoT-related reference architectures, frameworks, guidelines, platforms and standards. For IoT vendors and original equipment manufacturers (OEMs), however, such evolving IoT landscape means bountiful amounts of both opportunities and risks. The same holds for the consumers who are going to buy such products. We present an IoT framework and protocol that is unconditionally forward compatible. Our work defines the minimal criteria for a device to qualify as an IoT-enabled device, which could be taken as reference by IoT OEMs to build their IoT devices accordingly, across varied applications and domains. Such knowledge could help them make an informed choice amongst various available target hardware. With this protocol, it is possible to generate user interface/s on the fly, as per the devices' functionalities.

This work is supported jointly by the CPDA funding and PRIUS grants from the Indian Institute of Technology Indore.

Research has been done with maiden name Chaitanya Krishna, publication was made with the (new) name Chaitanya Krishna Bommakanti.

✉ Gourinath Banda
gourinath@iiti.ac.in

Chaitanya Krishna Bommakanti
ee1200206@iiti.ac.in

Harsh Mohan
me1200315@iiti.ac.in

1 Computer Science and Engineering, Indian Institute of Technology Indore, Simrol, Indore, Madhya Pradesh, India

**Keywords** Internet of things · Reference architecture · Forward compatibility · IoT Protocol · Application layer · Security

## 1 Introduction

Cascaded successes of various information and communication technologies (ICT)—together with their reducing cost—and maturity in sensor and actuator technologies led to the birth of *Internet of Things* paradigm. The paradigm of Internet of Things (IoT) is taking automation and control to new heights. IoT is disrupting everyday living, manufacturing industries, defence systems, healthcare, etc. Estimates forecast the IoT device number—not counting mobile phones—crossing 8 Billion by the year 2020 [1]. Furthermore by 2032, Industrial Internet is poised to cross $10–$15 trillion worth [2].

IoT is all about connected systems via internet that can both provide real time data, and handle requests based on this real time data to make everyday systems act more intelligent. Here 'Things' are physical objects consisting of processor/s, sensor/s and/or actuator/s with some intelligence either in situ or ex situ [3]. Though at core, IoT is all about connected things and their access via internet, implementing IoT-based solutions is very challenging. The foremost challenges are due to the heterogeneity factor inherent with the IoT. There is heterogeneity in application domains, hardware and requirements from diverse perspectives. Due to this heterogeneity, IoT happens to be the only technology that has several definitions [3]. Besides this heterogeneity, since connectivity and energy consumption is involved and information processing opens venues for various sophisticated analytics [4], there are functional requirements based on rich analytics and non-functional requirements around security and minimal energy consumption and harvesting. Such challenges—due to heterogeneity, functional and nonfunctional requirements—can be addressed by defining IoT reference architectures, frameworks, guidelines, protocols and standards.

To address the challenges posed by the IoT paradigm, several consortia have comeup—with partners having expertise in various convergent technologies—and are working towards developing IoT reference architectures, frameworks and protocols. Some individual companies have released developer platforms and ecosystem based on certain reference architectures. For instance, ThingWorx is one such system released by PTC [5]. *Open Interconnect Consortium* (OIC) has recently released a reference IoT architecture [6] as well. The *Industrial Internet Reference Architecture* (IIRA) is a reference architecture for industrial IoT [7] catering for the manufacturing sector. Identifying the gravity of this drastically changing IoT paradigm, several nations have released their own IoT policies [8], and founded working groups involving their own national academic and industrial experts in the related and relevant ICT areas. Several efforts means several non-unified frameworks. Consequently, several reference frameworks would only mean postponement of heterogeneity due to multiple disjoint frameworks.

In this dynamic IoT landscape, no unified protocol exists to support for diverse IoT-based devices [9]. For this very reason, several IoT device OEMS—standing at the crossroads of evolving protocols and frameworks—have bountiful options that might be opportunities or risks, which can only be decided by time. The current situation of the IoT device OEMs is similar to that of the DVD-vendors during the *Bluray Vs. HD-DVD war*.[1] On the other receiving end are IoT device consumers or end users with dilemma about which IoT-enabled device to buy. Again, the situation of these IoT consumers is comparable to that of the DVD discs' and players' consumers during the same period. Moreover, majority of these upcoming frameworks are trying to address several issues in a single go, which from an OEM perspective is too much to deliver from one single OEM. It is always good to have a robust ecosystem, but then this requires pacts between various vendors that is wishful thinking.

Therefore, in this paper, we present an operational framework and an affiliated protocol which could be a guideline for IoT-related OEMs on how to make their devices qualify as IoT-enabled devices. This protocol allows discovery, advertisement, invocation, and configuration of IoT devices. Such steps would allow the protocol to adapt to many scenarios where it may find an application, as the protocol has the ability to learn any device's capabilities on the fly which is done through a 'probing' mechanism as described later. The protocol, though caters human-in-the-loop kind of interaction, is fairly usable for Machine to Machine (M2M) interaction as well. We also describe the end to end message format used in the communication. This protocol addresses some security hurdles faced in the IoT paradigm by its integrated approach. We then demonstrate a system architecture that may be used on an IoT device that leverages our protocol, to create full-fledged IoT ecosystems and take support of existing IoT ecosystems with sufficient plumbing in place.

[1] In mid 2000s, there were two competing high-definition DVD video formats: Bluray and HD-DVD. Bluray was advocated by the group led by Sony Corp.; while HD-DVD was from the other group led by Toshiba Corp. In the end, Bluray was adapted by majority. Though Sony did benefit by Bluray, it is well known how bad the DVD market suffered due to the advent of streaming media.

## 2 Background

In this section, we will discuss a few preliminary concepts that forms basis for our framework and protocol. Here we also give our definition of IoT.

### Embedded system

Embedded systems are microcontroller-based electronic control systems together with one or more sensors and/or actuators. They are often embedded in a larger entity to control/monitor their operation. This *embedding entity* serves the main functionality to the end user. For example, in a smart air conditioner, air conditioner (AC) is the entity and the smart controller is the embedded system. Here the end function of AC is to condition the air as per user requested temperature. The embedded system comprises: microcontroller executing the control law; thermometer and hygrometer as sensors to read temperature and humidity, respectively, and control switches to cooling system as the actuators.

*Sensors and actuators*

Sensors and actuators are the bridging links between digital and physical worlds. They are the end interfaces that make it possible to access status and control operations of IoT-enabled devices. Sensors detect changes in measurable quantities of the environment around it, and relay this information back as a signal. Actuators on being activated with a signal generate a physical stimulus in its environment.

### 2.1 Internet of things

Inspired by definitions [10,11], we define IoT paradigm as following.

**Definition 1** (Internet of Things) A world wide network of interconnected embedded devices that are natively (Internet Protocol) IP-enabled, hence interconnectable and uniquely addressable based on standard communication protocols. It

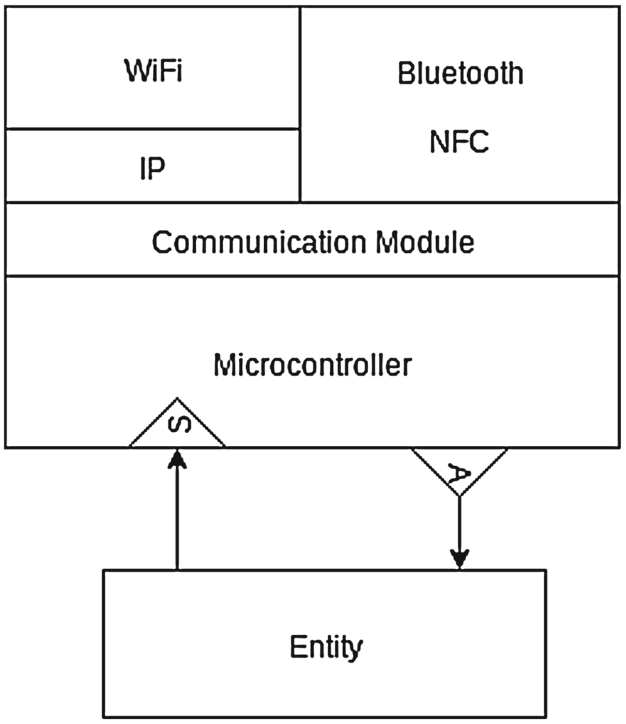


**Fig. 1** IoT node architecture

also includes internet services monitoring and controlling those devices.

The IoT architecture that we consider comprises four components:

1. IoT node/s
2. Internet connectivity layer
3. Cloud layer and
4. User Agents.

**Definition 2** (*IoT node*) IoT node is an embedded device comprising a microcontroller with IP network interface and sensors and/or actuator/s. It also includes the embedding entity.

If a device qualifies to be an IoT node, we call it as an *IoT-enabled device* or a *Thing*. Figure 1 summarises the composition of an IoT device. Since the interface between the embedded device and the embedding entity is internal to an IoT node, its upto the sole discretion of OEMs on how they implement it, and hence is of no interest to this paper. However, the vendor of the IoT device shall provide the IP connectivity interface. The example for this could be standard ethernet interface, a GSM-based interface, etc.

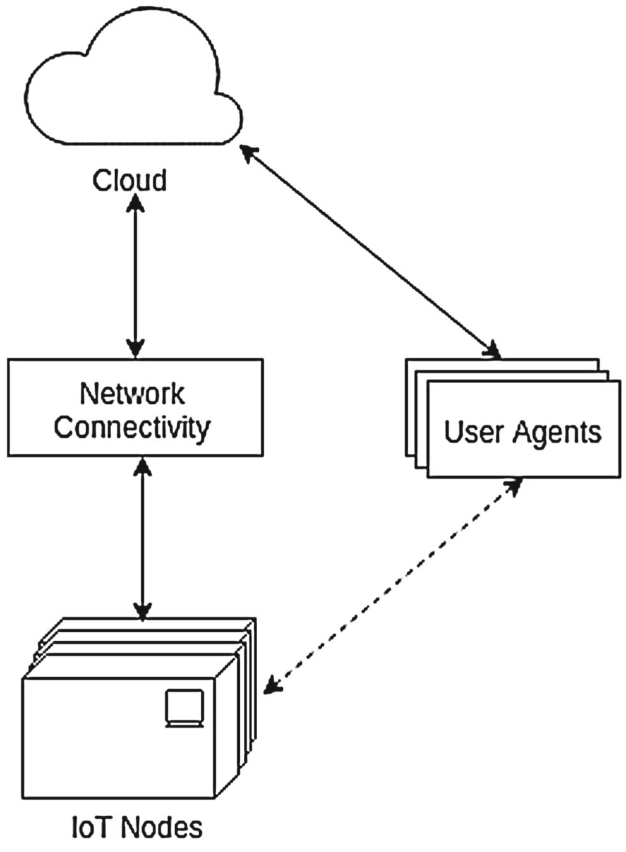


**Fig. 2** IoT architecture

### 2.2 Minimum framework

There are a wide variety of definitions for IoT paradigm as reported in [3]. But in this paper we give a minimum framework that leaves room to develop IoT-based applications as per any of those definitions using appropriate plugins and additional layers.

Our framework includes: remote-host/s (on Cloud), connectivity layer (both wired and other wireless modes) and IoT nodes. Figure 2 gives our IoT framework architecture. This means any IoT device's OEM shall provide:

1. to its end user the kind of remote-host service we present in our later sections and
2. the IoT node with certain minimum behaviour which is detailed below.

### 2.3 Minimum behaviour from an IoT node

The minimum behaviour expected of an IoT node essentially includes:

1. receiving the commands from authorised remote host/s and user agent/s (via wired and/or wireless media)

2. execute the commands accordingly and
3. periodically reporting operational status and sensor information to authorised remote host/s.

The proposed IoT protocol specifies this minimum behaviour in detail. Besides the earlier discussed minimum behaviour, depending on the type of application and capabilities of the entity and microcontroller, an IoT node could have additional intelligence in situ and/or ex situ. It is always encouraged to have minimum in situ intelligence to take care of abnormal situations, such as on internet outage, detection of attacks such as denial of service (DoS) and distributed denial of service (DDoS).

### 2.4 Protocol entities

The proposed protocol operates on the application layer and is theoretically communication medium independent. This protocol involves three entities: an IoT device, *User Agent* (Client), and Cloud. User agent is a client interface here for the IoT device users. Smartphones could be user agents with an appropriate app or web browser as the client interface. We developed an app for Android-based phones which generates an User interface on the fly to control the IoT device.

The Client is essentially a host machine that allows remote access to IoT devices using various communication media including, but not limited to Wi-Fi and Bluetooth. Such client typically functions as a dashboard for end users to manage their IoT device. However, same clients may exist as bridge nodes as well to support inter-device communication from the point of view of our protocol. This is discussed in Sect. 9.

A Cloud, on which the remote-host is running, is a necessity here to keep track of large number of IoT devices and their mapping to the authorised users. Since our protocol could be integrated into any other ecosystem, such extensibility could be easily achieved via cloud through this host. To use the protocol via the Internet, a Cloud network first provides user accounts to help users keep track of their IoT devices, and to grant or revoke access to IoT devices owned by them. All such users who have access rights of an IoT device are called authorised users. Further, the cloud also makes possible seamless communication to IoT devices which may not be on a static IP address. This is detailed in Sect. 4.

## 3 Protocol

In this section, we will go through the proposed protocol. As mentioned before, the protocol will operate on three entities: the Cloud, Client, and IoT-enabled device itself. To elucidate core aspects of the protocol, we will demonstrate some of the communication flows that are typically encountered in an IoT paradigm. The next subsections will describe these communication flows:

– *Discovery* and *Registration* of IoT-enabled devices and
– *Advertisement* of a Thing's capabilities.

Post these prerequisite communications, it becomes possible to operate the IoT device. The following are the communication flows corresponding to operations on the IoT device:

– *Invocation* of a Thing's operations and
– *Configuration* changes of the Thing can be carried out.

This protocol is in the application layer. By abstracting various communication interfaces such as Bluetooth/NFC/USB with appropriate wrappers, we can achieve uniform communication abilities. Such capabilities are highly desirable, for when in range, IoT devices could be accessed via the Bluetooth/NFC/USB interfaces. Such non-IP-based communication also means bandwidth conservation. Furthermore, in situations when there is internet outage, such direct communication becomes very essential.

### 3.1 Discovery

Before any communication can actually take place, discovery of available nodes to communicate with is a vital part of any protocol. Therefore, the DISCOVERY method performs a local scan via Wi-Fi, Ethernet over LAN, Bluetooth and other short ranged protocols. This discovery phase would not be applicable for IP-based communication. The discovery involving IP-based communication is explained through the next phase. Further, discovery can be overridden with local settings, if the IoT device has been chosen to be invisible by the respective owner.

### 3.2 Registration

Typically, we can communicate with an IoT device once we have its IP address and key. The concept of key will be explained later. However, if the device is not on a static IP address, it becomes mandatory to keep track of its current IP address, which is not a trivial task. Our protocol addresses this problem as well, as described in Sect. 4. To achieve this securely, on Cloud corresponding web host maintains a many-to-many relationship between Things and their authorised Users. To add a new Thing-User relationship to this mapping, we have the 'REGISTER' request. Whenever a user wants to add a new Thing in his/her Thing account (held on the cloud), he/she has to send his access credentials, Thing-ID and Thing-key. Figure 3 illustrates the sequence during Registration.

**Definition 3** (*Thing-ID*) Thing-ID is a unique identifier provided by the vendor of the IoT device.

This Thing-ID is something similar to EPC[12].

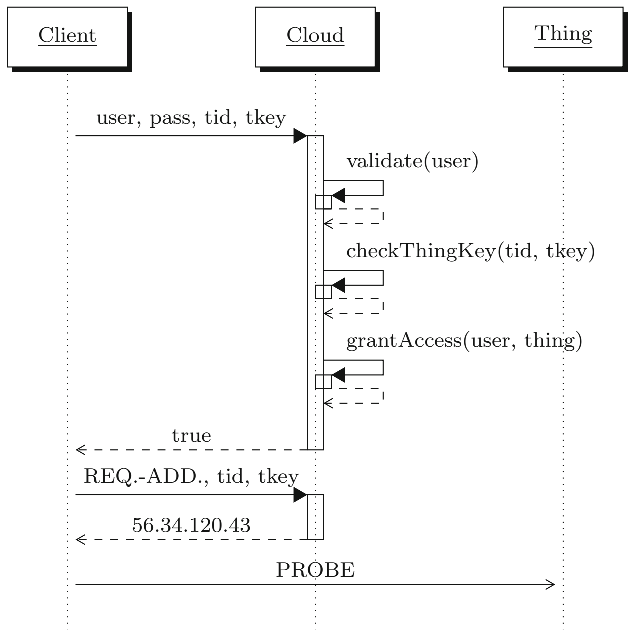


**Fig. 3** Registration flow

**Definition 4** (*Thing-Key*) Thing-Key is a private secret code of sufficient length.

In this paper, we consider a key of length 128-bits. This key shall be provided by the OEMs. It is conceptually similar to bank users receiving a pincode for their debit and/or credit cards.

Cloud, through a running instance of a web host, would then first validate his/her credentials, check if Thing-ID and Thing-Key match, and if they do, grants him/her access to that particular Thing. By doing this, Client is now able to send 'REQUEST-ADDRESS' requests whenever it needs the IP address of the Thing, making remote communications to device possible even when its IP changes.

## 3.3 Advertisement

After a successful Registration, the user is authorised to invoke a Thing. However, before invoking an operation, the Client (a user agent) must be aware of:

1. The operations offered by the given *IoT device* and
2. The expected way to invoke any specific operation on that *IoT device*.

To know this information about a specific thing, a client sends out a 'PROBE' request and expects a parseable manifest in response (See Fig. 4), typically in an XML or a JSON format. As can be seen in the Listing 1, there could be several `operation` tags corresponding to respective operations offered by the device. The format must follow a specific

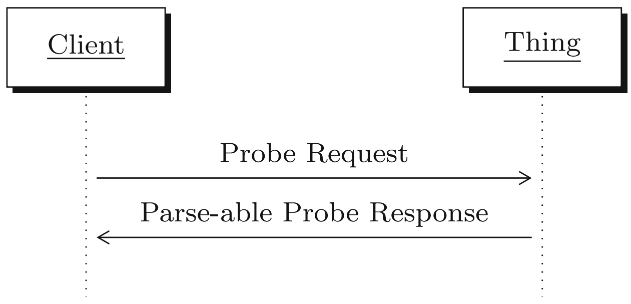


**Fig. 4** Advertisement flow

predefined template to include the details, such as operations offered, operation invocation identifiers, expected I/O format for each operation and even a small human readable description of each service with optional documentation. This also makes possible to generate user interfaces dynamically *on the fly* to make intuitive dashboards specific to each Thing. An important feature worth noting is that advertisement allows the device to be over-the-air *update friendly*, as new (soft) operations can be invoked without changing the client or device.

**Listing 1** Example XML Probe Response

```xml
<operations>
    <operation name="SetLightColor"
        >
        <description>
            Change the light color.
        </description>
        <input name="Red">
        <input name="Green">
        <input name="Blue">
        <output name="Status">
    </operation>
    <operation name="GetStatus">
        <description>
            Know whether light is
               switched on or off.
        </description>
        <output name="Status">
    </operation>
</operations>
```

## 3.4 Invocation

After an IoT device has been registered, and its advertised manifest has been parsed, it can be used like a normal IoT device, and its operations may be invoked remotely. The advertisement flow reveals a comprehensive device manifest that is parsed and dynamic interfaces are generated on the go. Documentation supplied about each operation should help elucidate essential aspects about that operation and how to invoke that operation, if not readable already. This way operations/services of a thing can be invoked using the 'INVOKE' method of the protocol, along with a service identifier, and input if any, and the outcome can be displayed on an inter-

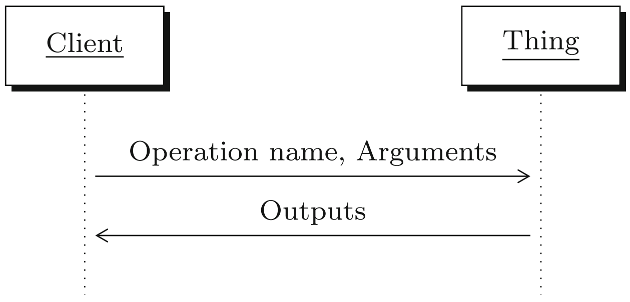


**Fig. 5** Invocation flow

face built atop the output format taken from the manifest, thus allowing for consumer friendly IoT implementations. Figure 5 illustrates the invocation flow.

### 3.5 Configuration

Another use-case for IoT devices includes *configuration updates*: changes in protocol variables and server parameters. Configuration updates that change core functionality of IoT devices should be addressed by appropriate Invocation requests, supplied by the manufacturer. Other configurations that are generic and not implementation specific, are addressed by the 'CONFIGURE' method of the protocol. These changes include changes to *tick rates*, *security parameters* and other *protocol variables*. These protocol variables are explained later. The main reason these variables should be altered is to improve performance metrics of these devices relative to their use case. Since a wide array of IoT devices would take over the technology market soon, with applications of varying uses, each implementation would have their own 'sweet spot' metrics, with some implementations not being time critical, and thus being conservative on their power requirements, while others being extremely time critical at the cost of a little extra power consumption. The CONFIGURE method exactly serves for this cause.

## 4 Communication media

As mentioned in Sect. 3, the protocol is designed to work on any communication medium. However, the protocol needs to be tweaked for smooth performance in certain communication media. More specifically, we differentiate between communication medium on the basis of the addresses used to facilitate communication. In doing so, we classify media into two classes:

– *Static Addressed Media*, or those media which operate on static addresses, or hardware addresses like bluetooth [13], NFC [14] including *static IP address* and
– *Dynamic Addressed Media*, including those which operate on addresses that are subject to change, such as the Internet which operates on dynamic IP addresses assigned by *Internet Service Providers* (ISP).

Since the Internet works on IP addresses, which are typically given on lease, they do not provide static addresses to services on the web. Static addresses and routes are expensive, and with the 8 Billion devices that are going to take over consumer electronics by 2020, this is clearly not an option for the general public. Therefore, to address this issue, we use the Cloud as static reference on the internet. Each IoT device that uses our protocol would periodically 'ping' the cloud when given Internet access, therefore, giving the cloud a constant visibility of the IoT device. This is done using the 'UPDATE-ADDRESS' method of the protocol. The rate of pinging that is the tick rate is configurable as stated earlier in Sect. 3. The higher the real time demand of services of IoT devices are, the higher would be the tick rate. The cloud is hardwired to listen to such 'heartbeats' from Things, and updates its local cache with the last known address. This address is used as a fallback in the event of an IP address change. The Client itself stores a local copy of the last known address, only this cache might go stale faster as the Client is also dynamically addressed, and thus making it impossible for the thing to notify the client directly. Thus the client acts as a 'bridge', as stated above.

To reiterate, the Client cache will always go stale faster than the Cloud cache. On the event of an *IP address change*, the Cloud gets notified of the change after the next 'heartbeat'. The client, however, might still operate on the stale cache, and any attempts to invoke a service results in: (i) a timeout, because no device currently holds the address, or (ii) a protocol failure or an authorisation failure on the event that another device takes up the old IP, but either fails to understand the protocol we use or understands the protocol and cannot authorise the request due to a key mismatch. Both these cases require a call for a cache update on the client side, and the cloud uses the 'REQUEST-ADDRESS' method of the protocol to update its cache, and retry the request. This mechanism is depicted in the sequence diagram shown in Fig. 6.

## 5 Security

In this section, we will look at some of the security aspects of the protocol that would otherwise leave it vulnerable to exploitation. We are inclined to a security policy that promises safety and availability. Security is a big concern [15, 16] for the IoT paradigm in general. Our protocol covers security from the perspective[2] of IoT device security and associ-

[2] The perspectives for security include: network security, server security, data security

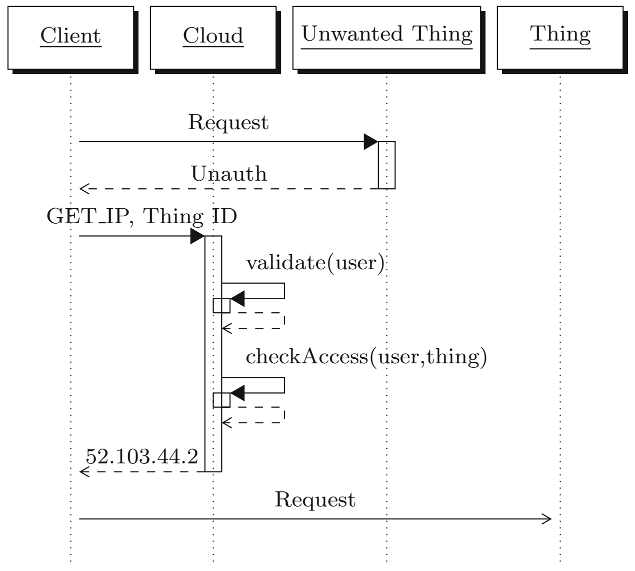


**Fig. 6** Handling IP changes in DAMs

ated data security. IoT paradigm essentially involves web-based abstractions and/or interfaces for physical devices, exploitation of these devices would not only cause the loss of data, but also the loss of fidelity of the associated physical systems [15]. When the IoT device numbers are in the order of billions [1], it results in chaos. Therefore, security should be natively built into IoT-based solutions [16], and thus IoT solutions should offer at least a level of robustness that could be affordable from the available processing power of the IoT device. Since the processing power with an IoT device can be limited, security can also be provided as a service from cloud. Our protocol provides security that is affordable with IoT devices. Besides constrained processing power, affordability is defined by the available energy and bandwidth.

In the proposed protocol, we have strong urge and need to encrypt 'raw' packet data from end to end, with accepted encryption standards of sufficient encryption entropy. The only fields worth exposing in plain text include a header describing the protocol name and version, along with the encryption type used. As mentioned in Sect. 4, communication is only possible: post successful registration that would distribute an authorisation key, post registration for communication. The authorisation key itself could be the device key, or an OAuth[3] like key, generated by the cloud which can be revoked as and when needed, to better support multi-client ecosystems and is recommended. Encryption must be performed whenever data is exchanged between two entities, and for maximum protection, a different key must be used for each entity pair.

[3] OAuth provides applications a 'secure delegated' access for service owners to authorise third party access to their resources

Another key security aspect to consider whilst designing a secure packet, is the presence of a salt [17], to further strengthen the encryption which is inevitable in the IoT ecosystem. Consider the scenario where an attacker taps into a private network and sniffs encrypted packets used to communicate with IoT devices. Considering a saltless encryption, the packets used would lack sufficient entropy, as the cipher-text of any invocation packet would not change as long as the plain text or the actual request itself remains the same. This would allow the attacker to correlate the encrypted packets to the physical outcomes performed by the IoT devices either manually or by deployment of Trojan sensors in situ. Either way, the outcomes would be cracked without the need of cracking the encryption itself, thus nullifying the complete encryption process. A salt that would consist of a well generated random number would help make the encryption much stronger by increasing the entropy of the cipher-text by multiple orders of magnitude.

Another attack vector for IoT devices are Dos and/or DDoS attacks [18,19], because of their taxing nature even on full-fledged servers, DDoS prevention at IoT device level is difficult. However, DDoS attacks can still be countered by rerouting requests to a cloud with one of the several existing state-of-the-art DDoS filter services to prevent direct attack mechanisms on smaller embedded IoT devices, which would otherwise cripple because of their limited hardware capacity. One way to achieve this would be to implement minimal logging support on the IoT device which would report failed attempts at the communication step, and the source IP address to the cloud that would then log these failed attempts, and relay these to a DDoS detection filter in real time, which would then analyse the logged requests with recent history of logs from the same source IP address (or a range of certain IP addresses), and classify the request as genuine or malicious, and accordingly relay instructions to the IoT device to block incoming connections from the malicious source. [20] proposes a technique to identify DDoS attacks.

## 6 Message structures

After elaborating on the protocol itself, we will now substantiate on how exactly the protocol will actually be transmitted on the network. In this section, we will establish the application level packet structures and byte sizes that are sent on the wire, taking into consideration all the proposals made so far. As mentioned earlier, security would be a vital part of the protocol, thus most parts of the messages would be encrypted leaving a small unencrypted header which would identify the protocol name, version, and the encryption type used. The *Response* message would contain an additional unencrypted header *Status*. Also, the encryption would be

| | |
|---|---|
| Protocol Name &Ver | Encryption Type |
| Payload | |

**Fig. 7** Request/response message structure

| |
|---|
| DISCOVER |

**Fig. 8** Discover request payload

| | | |
|---|---|---|
| Thing-ID | Thing-Key | Thing-Type |

**Fig. 9** Discover response payload

| | | |
|---|---|---|
| REGISTER | Thing-ID | Thing-Key |

**Fig. 10** Registration request payload

salted, to increase the entropy of the resulting digest and making it harder to crack, as explained in Sect. 5.

These common headers to all message makes up the top level packet and is structured as show in Fig. 7. The fields *Protocol Name*, *Protocol Name* and *Encryption Type* take up 2 bytes each. Whereas the size of the field *Payload* depends of type of message being sent. The payloads for each type of message and their respective responses are as enlisted in the next few subsections. Irrespective of the message type, it is essential to note that particular payload would always sit inside this top level packet, and will always be sent on the wire in this format.

## 6.1 Payload structures

The payload structure for each of the type of message allowed in our protocol as described in the Sects. 3 and 4, are concertised in the below subsections. Figures 8, 9, 10, 11, 12, 13, 14, 15, 16 and 17 illustrate various payloads explained in the following.

### *6.1.1 Discovery payload*

The discovery payload is the only payload which is left unencrypted. It includes the data shared during discovery that is the Thing-ID, Thing Name, and Thing-Type.

| |
|---|
| Authorisation Keys |

**Fig. 11** Registration response payload

| |
|---|
| PROBE |

**Fig. 12** Advertisement request payload

| |
|---|
| XML/JSON/Other Parse-able Format |

**Fig. 13** Advertisement response payload

| | | |
|---|---|---|
| INVOKE | Operation Name | Arguments |

**Fig. 14** Invocation request payload

| |
|---|
| Outputs |

**Fig. 15** Invocation response payload

### *6.1.2 Registration payload*

Registration payload contains information that is required during the registration process: including the Thing-ID, and Key combo in the Request, and Authorisation Keys if any in the response.

### *6.1.3 Advertisement payload*

Advertisement payload contains the data shared during the advertisement phase that is only the parseable format in the response.

The request payload contains a single byte which identifies that the message is of the type *PROBE*.

The response payload will contain the details of metadata of the thing and operations it provides in a parseable format, which could be a verbose string format like XML/JSON or a more compact format like BSON.

### *6.1.4 Invocation Payload*

Invocation Payload contains information necessary to handle a specific service/operation. The request includes *Invocation token* along with the required inputs, and the response includes corresponding outputs.

| | | | |
|---|---|---|---|
| CONFIG | Param name 1 | Param Value 1 | .. |

**Fig. 16** Invocation request payload

| | | | | |
|---|---|---|---|---|
| Param name 1 | Param Value 1 | Param Name 2 | Param Value 2 | .. |

**Fig. 17** Invocation response payload

The first byte of the request payload identifies that the message is of the type *INVOKE*. The next byte of the payload represents the size of the string *Operation Name*, the actual bytes of which would follow next. The next byte represents the number of arguments following, each of which contain one byte representing the data size of the argument and the argument itself.

The response is formatted in a similar fashion, with each output having the first byte representing the size of output followed by the byte data of the actual output.

#### *6.1.5 Configuration payload*

Configuration payload has configurations information, and consists of param-name and param-value pairs in the request payload, while the response would contain the final configuration of the Thing.

The first byte of the request payload identifies that the message is of the type *CONFIG*. The next byte of the payload represents the number of config changes being requested, with the actual requests following next. Each of the config requests contain a key-value pair: both of which begins with one byte representing the data size of the key/value, followed by those many bytes of the key/value itself.

The response is formatted in a similar fashion, with each key-value pair representing the new, updated config data.

#### *6.1.6 Heartbeat packet*

The Heartbeat packet should be encrypted. In the event of an incorrect key, the encryption algorithm fails. Figures 18 and 19 gives the Heartbeat request and response packets' structures.

#### *6.1.7 New address payload*

New address payload operates on the device ID, and would be authenticated by the user session/registration status on the cloud. Figures 20 and 21 show the corresponding payloads.

| | |
|---|---|
| Protocol Name &Ver | Encryption Type |
| UPDATE-ADDRESS | Unencrypted Device ID |
| Encrypted Salt | |

**Fig. 18** Heartbeat request packet

| |
|---|
| ACK/NACK |

**Fig. 19** Heartbeat response payload

| | |
|---|---|
| GET-ADDRESS | Thing-ID |

**Fig. 20** New address request payload

| | |
|---|---|
| Device ID | Device Address |

**Fig. 21** New address response payload

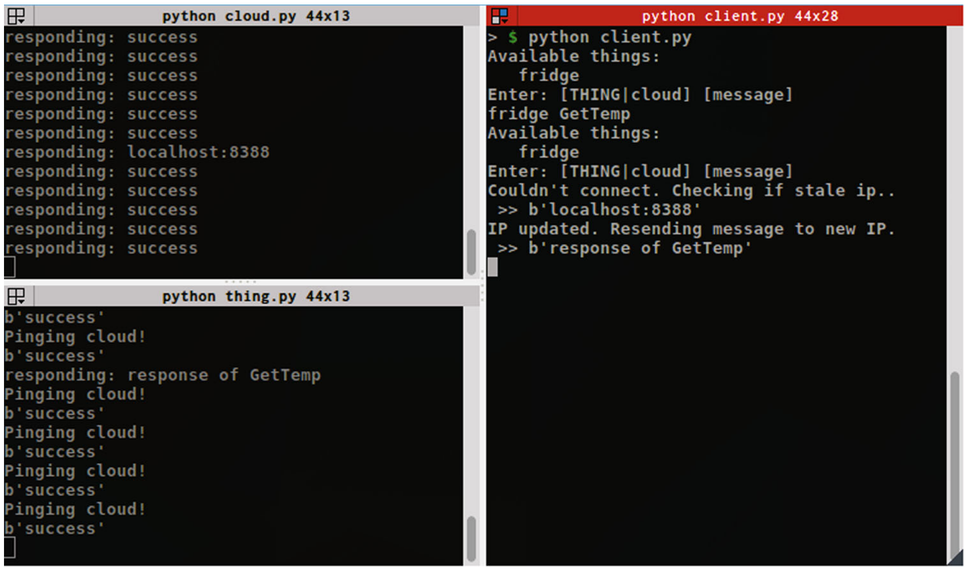
## 7 Experimentation

### 7.1 Basic simulation: to analyse communication flows

The objective of the first experiment we conducted was to test the integrity of various communication flows and the overall behaviour of the protocol. For this, we planned to carry out the experiment with a handful of commonly used sample Things. Also, since we are not concerned about any potential hardware specific problems of Things, we experimented with simulated models of Things instead of real hardware. Each thing is modelled as a process. The process models phases of the Things' operation cycle: *initialisation step*, *functionality step*, *interaction step* and *termination step*, corresponding to: booting of the IoT device, delivering the functionality, waiting & listening for the client requests and shutdown, respectively. We modelled various types of Things, such as Microwave, Washing Machine, Fridge, along with that of an instance one each of Client and Cloud. Thus, the simulation setup consisted of a set of processes running on different computers connected via internet. On such a simulation setup, we tested various communication flows corresponding to various use-cases and scenarios.

Our experimental simulation setup consisted of multiple computers, one corresponding to a particular Thing acting as an IoT node, another acting as a Client. Each of the Things and Client were written in *Python* [21] using the *Twisted framework* [22], and are essentially processes able to communicate through the Internet. For Cloud node, we deployed it on an Amazon Web Service's Elastic Compute Cloud (AWS EC2) instance, making it run on a static IP address. Thus, all in all we had three different types of processes corresponding to things, clients and cloud in our framework, each running on a different host.

We initially experimented with the IP Handling flow described earlier, on a single machine by running three different processes to emulate each of the following: the Cloud, a Client and a Thing. The Cloud and the Thing simulations act as servers, with the Cloud always listening on a fixed port. The Thing, however, listening on a random port which changes periodically. This is to simulate IP address changes / reallocations which occur on the Internet. The Client process works like a shell, allowing commands of the form (target, opcode), which would relay messages to the target address



```
python cloud.py 44x13
responding: success
responding: success
responding: success
responding: success
responding: success
responding: success
responding: localhost:8388
responding: success
responding: success
responding: success
responding: success
responding: success
```

```
python thing.py 44x13
b'success'
Pinging cloud!
b'success'
responding: response of GetTemp
Pinging cloud!
b'success'
Pinging cloud!
b'success'
Pinging cloud!
b'success'
Pinging cloud!
b'success'
```

```
python client.py 44x28
> $ python client.py
Available things:
    fridge
Enter: [THING|cloud] [message]
fridge GetTemp
Available things:
    fridge
Enter: [THING|cloud] [message]
Couldn't connect. Checking if stale ip..
 >> b'localhost:8388'
IP updated. Resending message to new IP.
 >> b'response of GetTemp'
```

**Fig. 22** Three process simulation on single machine

stored in cache locally. If the address lease of the target has expired, then the request would result in a timeout, protocol mismatch, or authorisation failure, as explained in earlier sections. This is followed by a 'GETADDRESS' request to the cloud to get the new address (if address has changed), and request the same service on the new address.

Figure 22, for example, shows one of such experiments where IoT node used is a model of Fridge, and the communication flow which is being tested is the IP Handling flow. The image shows the simulation corresponding to the scenario when where the Fridge had an IP address change from some earlier address to http://localhost:8388. It can be seen only when the Client tries to connect to Fridge. It first fails as it tries to connect to the earlier IP address, but then realises this and sends a request to the cloud to get the new address being used by the Fridge. Cloud has the knowledge of the current IP address of the Fridge because of the periodic heartbeat sent by the Fridge, and hence responds with this information to the Client. With the new address available, Client now sends the request again on the new address, this time succeeding to connect to it. This confirms the expected operation of our protocol.

## 7.2 Example client (android application)

To demonstrate a real life and consumer friendly application of the protocol, we have made an android application (screenshots of this App can be seen in Fig. 23) which operates on our protocol, and provides an interface to manage IoT devices online. Each screenshot corresponds to different steps in the user interaction with the IoT device/s. Application operates on the cloud authentication, prompting a login on first use. The top row leftmost screenshot corresponds to this login and authentication. This generates a list of registered devices, whose services may be invoked. The second screenshot in the top row corresponds to post successful login where the user is shown the list of IoT devices over which s/he has access rights. An IoT device's services list is populated using 'PROBE' method of the protocol, and the supplied manifest that follows. This is seen in the top row rightmost screenshot. All in all, the client attempts to deliver an intuitive way to interact with IoT devices using the proposed protocol. It is intuitive because the user interface is generated on the fly based on the response to PROBE-request. For the air conditioner kind of IoT device, we can access specific operation such as fan speed and temperature set point. The screenshots in the bottom row correspond to this interaction. This android app also lets the users to login to their things' collection, recall that we had in the protocol a step where user-thing mapping is established. The android app, after successful login, gives a view of this mapping.

## 7.3 IoT hardware setup

The goal of the final experiment is to replicate the exact scenario of how the end deliverables might be actually used. For this, we had run the proposed protocol on the actual hardware devices. Our setup consists of one Client node, one IoT node and a Cloud. The Client node is an android application we developed which includes the implementation of protocol, specific to a client node. The android phones, which we used for testing purposes include Samsung GT-N7000 and Samsung GT-i9300, both running Android 6.0.1. The IoT node is a smart bulb, which is a configurable LED with certain functionality. This LED bulb can be configured regarding its blinking frequency, colour of the light from LED via changing the Red, Green and Blue component values, etc. Furthermore, it can report whether it is turned ON or OFF.

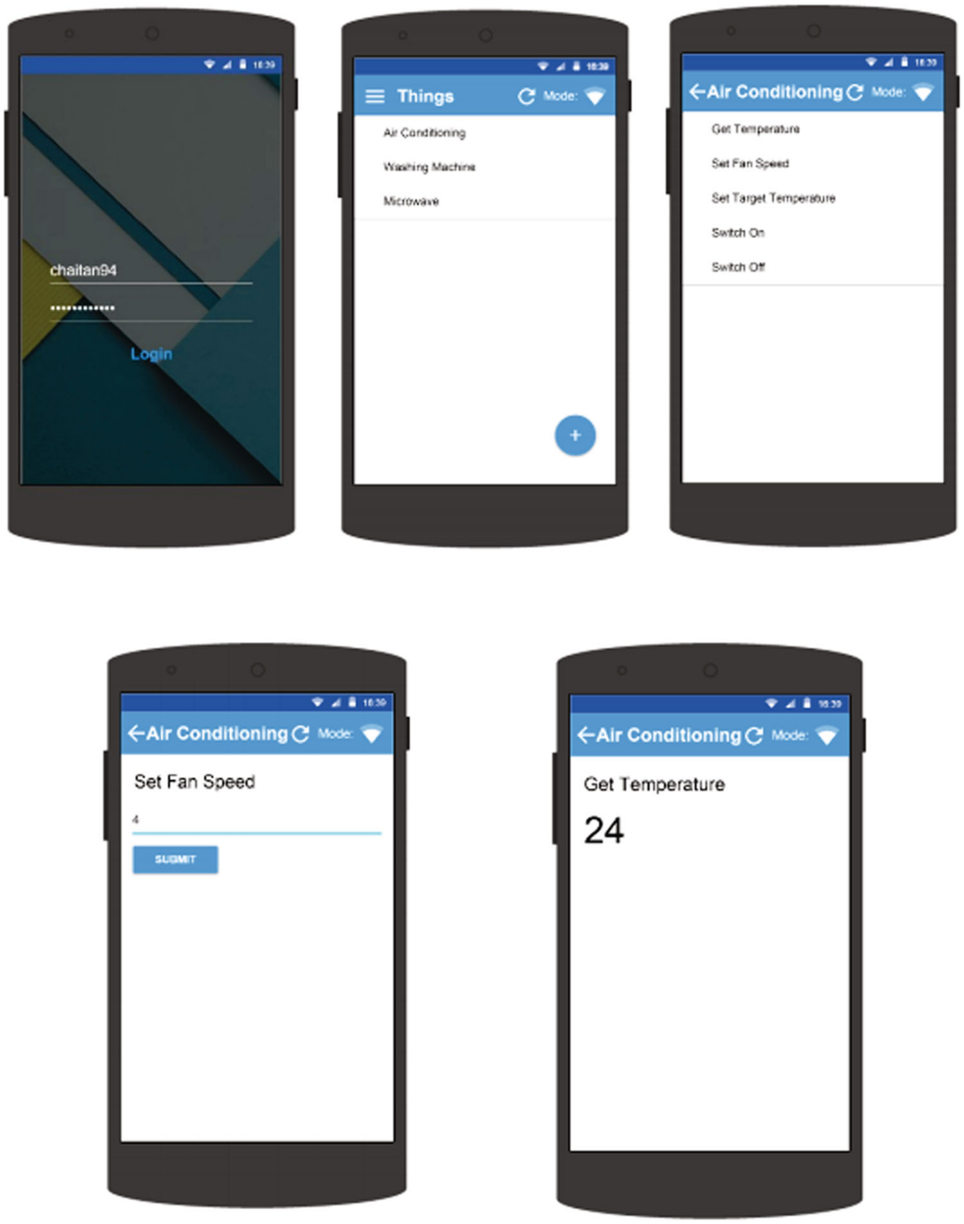


**Fig. 23** An example android-based client

The thing interface for this IoT node included *Intel Edison* with an x86 microprocessor architecture running *Yocto Linux*. This board consisted of a python (v2.7) implementation of the framework. The experimental setup can be seen in Fig. 25. This frameworks thing server abstracts bluetooth and Wi-Fi servers, which accept packets and relay them to underlying modules. The received packets are then checked for validity, failing which results in an immediate drop of the packet. Valid packets are then parsed and routed to the appropriate request handlers (See Fig. 24). Once a request is serviced, we generate response based on the result of the respective handler, and convert it into a packet, and relay the response packet. This also represents the minimal framework required at the IoT device to operate on the protocol.

## 8 Related work

The work in the field of IoT has been very diverse. There are a multitude of protocols being developed by various research groups, both from industry and academia. Constrained Applications Protocol (CoAP) [23] is one such protocol that has gained broad popularity. There is also a software framework, being developed by the AllSeen Alliance, known as the *AllJoyn Framework* [24]. Although started by Qual-

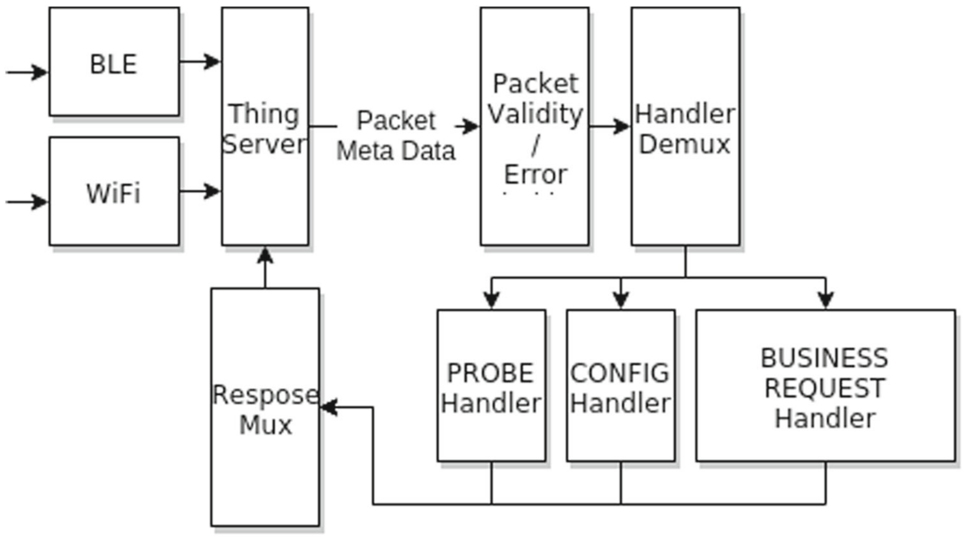


**Fig. 24** Thing's software architecture in our hardware setup

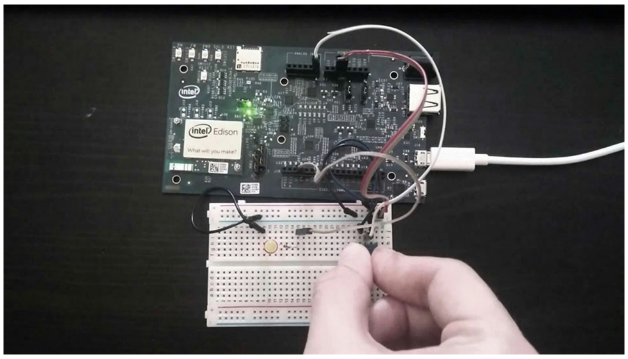


**Fig. 25** LED IoT node using an Intel Edison

comm, many established organisations are now collaborating towards it, including companies such as Cisco, Sony, LG, Microsoft and Panasonic. Currently AllJoyns source code has been signed over to the Linux Foundation and is under a very active development. There are also many proprietary organisations contributing to the field of IoT. Wigwag [25] is one such IoT Platform. There exists dense collection of IoT frameworks [26], platforms [5] and standards [27] proposed by many consortia which can be consulted in [3]. Majority of them have no flexibility and forward compatibility as offered by our protocol. One example for flexibility in our protocol is generation of user interface (at the client side) on the fly depending on the operations defined in the manifest. The forward compatibility of our protocol is due to the abstraction defined in our framework. No matter what the technology landscape might look like, it always consists of the entities—in one form or the other—as defined in our framework. This brings in the unconditional forward compatibility of the IoT devices, which are implemented as per our framework and protocol.

RFC 7252 [23] defines CoAP as "The Constrained Application Protocol (CoAP) is a specialised web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things. The protocol is designed for M2M applications such as smart energy and building automation...". CoAP is heavily inspired by the simplicity and versatility of representational state transfer (REST) style of the usage of HTTP. While being consistent to RESTful principles, CoAP also adds specialised features like support to multicast messages. Though such multicast feature is useful in group communications, it has poor reliability. For this reason, it exploits unicast via an intermediate agent that interacts with the members of a group. Like our protocol the security is via key-based encryption.

The AllJoyn protocol is in many ways similar to our protocol. Both our proposed protocol and AllJoyn have an advertisement and discovery phase. Both are independent of, the programming language used, operating system of the device and transport layer of the communication medium. M2M is made possible via event-driven actions and finally both of the protocols implement security measures at the application level. There are, however, major differences between our proposed protocol and AllJoyn. AllJoyn divides the participating agents of the protocol into two types: AllJoyn App and an AllJoyn Router. Whereas, in our protocol we have three categories of participating devices: IoT node, Client Node and a Cloud Node. AllJoyn App and an AllJoyn Router are in many ways analogous to IoT node and Client Node, however, by the introduction of Cloud Node in our protocol we are able to handle dynamic changes in IP addresses when devices move across networks and also gain easier and more control over security of the system by making the Cloud act as a key distribution center. AllJoyn also can handle changes in IP addresses of the devices, but through the usage of *Wi-Fi Hotspot 2.0* [28], which is as defined by the Wi-Fi Alliance.

## 9 Discussion and future work

This reported work could guide the OEMs to develop forward compatible IoT-enabled devices. The proposed protocol, which we named as *One IoT*, allows device discovery, advertisement, registration, invocation and configuration; while the framework includes authentication-host, internet, client/s and thing/s. We further address connectivity and security concerns in the protocol. While the protocol was designed for Human to Machine interaction, its simplicity makes it applicable and adaptable for Machine to Machine (M2M) communication as well. We are working on Version 2 of our One IoT protocol that works for M2M as well. This could be done by running scripts on Clients, or by making Broker Clients that act like smart hubs and help create smart IoT ecosystems. These scripts could be triggered synchronously (periodically) or asynchronously (event-based). However, identifying attacks and mitigation of those attacks are equally important. For attacks' identification and mitigation, our protocol being on the cloud, can plugin the state of the art attack filter services that become available on the cloud.

From a cultural perspective, application of IoT—irrespective of the domain of application—is going to positively disrupt both the people's ways of thinking (hence interacting and describing patterns) and the material objects that together shape a people's way of life. *IoT technology landscape* shall be such that this change happens in a seamless fashion. Our *One IoT* protocol is exactly an attempt in that pursuit.

### 9.1 Future work

The IoT paradigm makes it possible to implement a range of applications both in civilian and military domains. Furthermore, such IoT-based systems could be safety critical systems as well. The utilities industry is envisaging to realise unforeseen applications with IoT technology. No matter what the end application is and what kind of sophistication is made possible by the IoT paradigm, one should not forget that the Internet itself is the essential constituent of this technology. This means when an internet outage occurs, there is a risk that the IoT devices might carry out operations that might not be in the positive interest to the owners. We are working on the next version of One IoT protocol, which addresses this concern of internet outage. The solution we propose is on detecting an internet outage bring the IoT devices into a *no-internet-mode*. Such a mode is specific to the IoT device and the type of application domain. The idea is " whenever for an IoT device such a safe mode is defined, our protocol (on the Thing-side) would—on detecting the *internet outage* event—imposes the actuation controls bringing the device into that *safe mode*. We could achieve this behaviour in our protocol by having a method that actively reads the internet status. But again, the safe mode needs to be defined per device basis.

Furthermore, to facilitate: (a) training in IoT paradigm through our protocol and framework and (b) modelling and analysis of deployed IoT systems based on our protocol, we are developing a simulator for IoT deployments. This simulator envisages to make possible formal verification of *Quality of Services* in IoT paradigm both from qualitative and quantitative perspectives. The underlying verifier is from the work reported in [29].

## 10 Conclusion

We have seen how, through our protocol we are not only able to learn the capabilities of any generic IoT-enabled device, but also actually be able to utilise these functionalities through a generic client as well. With the advent of such a unified protocol, we fragment the IoT realm into two viscerally disjoint classes:

1. Hardware devices that react with the physical world, and
2. Clients that moderate the above-mentioned hardware.

We believe that this fragmentation is key to simplify and conquer the complexity inherent with the IoT paradigm. First, such a fragmentation allows efficient manufacture of these IoT solutions as both of these development phases are disparate, with one focusing on development of hardware specific implementations and device-specific logic development, or the backend of IoT, and the other on solutions to moderate and control a large number of such IoT devices, with possible emphasis laid on intuitiveness of their design, especially for end users—or the frontend for IoT. Further, this fragmentation would also pave way for the rise of small-time developers in the IoT paradigm allowing for domain specific innovations in fields like medicine, etc., and allowing each class to mature independently.

The advent of a universal protocol would also help IoT manufacturers in a plethora of ways, by reducing investment towards research and development of IoT protocols that reduce manufacturing costs and enable economic IoT implementations, and can also help systematise the manufacturing process. This allows OEMs to make their devices future proof and would allow interplay between IoT devices in the future, thus making these devices attractive from the point of view of end consumers. End consumers would also benefit from such unified protocols, as this allows wider variety, and thus pave way for personalised IoT ecosystems, which may also be tailored to user specific implementations.

## References

1. Harbor Research (2014) What exactly is the 'internet of things'?. http://harborresearch.com/wp-content/uploads/2014/03/Harbor-Postscapes-Infographic_March-2014.pdf
2. Evans PC, Annunziata M (2012) Industrial internet: pushing the boundaries of minds and machines. http://www.ge.com/sites/default/files/Industrial_Internet.pdf
3. IEEE: Towards a definition of the Internet of Things (IoT) (2015)
4. Bonomi F, Milito R, Zhu J, Addepalli S (2012) Fog computing and its role in the internet of things. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, pp 13–16. ACM, New York, NY, USA. doi:10.1145/2342509.2342513
5. Thingworx Platform. http://www.thingworx.com
6. Open Interconnect Consortium: The Open Interconnect Consortium and IoTivity (2015). http://openinterconnect.org/wp-content/uploads/2015/07/OIC-IoTivity_White-Paper_Final.pdf
7. Industrial Internet Consortium: Industrial internet reference architecture. Tech. rep., Industrial Internet Consortium (2015). http://www.iiconsortium.org/IIRA-1-7-ajs.pdf
8. Ministry of Communication and Information Technology India D.: Draft policy on internet of things. Tech. rep., Government of India (2015)
9. EU 2012 (2012) The artemis embedded computing systems initiative. http://www.artemis-ju.eu/
10. Information and Communications (ICT) Services businesses and technologies: Internet of Things Strategic Research Agenda (IoT-SRA) (2011)
11. Shelby Z, Bormann C (2009) 6lowpan—the wireless embedded internet
12. Thiesse F, Floerkemeier C, Harrison M, Michahelles F, Roduner C (2009) Technology, standards, and real-world deployments of the EPC network. IEEE Internet Comput 13(2):36
13. Bluetooth-SIG (2007) Bluetooth core specification version 2.1+ edr. Specification of the Bluetooth system
14. Baddeley D (1999) Identification cards–contactless integrated circuit (s) cards–proximity cards–part 2: radio frequency power and signal interface. ISO/IEC, pp 14443–2
15. Oltsik J (2014) The internet of things: a ciso and network security perspective. Tech. rep, Cisco Systems
16. Wind River (2015) Security in the internet of things. Tech. rep, Wind River Systems
17. Morris R, Thompson K (1979) Password security: a case history. Commun ACM 22(11):594–597
18. Mirković J., Prier G, Reiher P (2002) Attacking ddos at the source. In: 10th IEEE International Conference on Network Protocols, 2002. Proceedings. IEEE, pp. 312–321
19. Wood AD, Stankovic JA (2004) A taxonomy for denial-of-service attacks in wireless sensor networks. Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems, pp 739–763
20. Yogesh DS, Rajendra K, Neminath H (2015) An experience report on scalable implementation of ddos attack detection. In: Advanced Information Systems Engineering Workshops. Springer, Berlin, pp 518–529
21. Rossum GV (1995) Python Tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica
22. Kinder K (2005) Event-driven programming with twisted and python. Linux J 2005(131):6
23. IETF: The constrained application protocol (coap) (2014). https://tools.ietf.org/html/rfc7252
24. AllJoyn. https://allseenalliance.org/framework/documentation/learn. Accessed 21 Oct 2015
25. Wigwag IoT Platform. URL http://www.wigwag.com. Accessed 23 Sep 2015
26. Industrial Internet Consortium: Industrial internet reference architecture (2015). http://www.iiconsortium.org/IIRA-1-7-AJS.pdf
27. Open Internet Consortium: Reference implementation of the internet of things (2015). URL http://www.openinterconnect.org/developer-resources-specs.pdf
28. Orlandi B, Scahill F (2012) Wi-fi roaming–building on andsf and hotspot 2.0. Alcatel-Lucent and British Telecommunications, Tech. Rep
29. Banda G, Gallagher JP (2010) Constraint-based abstract semantics for temporal logic: A direct approach to design and implementation. In: Logic for Programming, Artificial Intelligence, and Reasoning—16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers, pp 27–45. doi:10.1007/978-3-642-17511-4-3