Final Report Research Engagement

On

IPv6 for IoT

NASSCOM CENTRE OF EXCELLENCE FOR INTERNET OF THINGS

Gurugram, India

24 January 2020

BACKGROUND

The association between NASSCOM CoE-IoT and ICANN started in the November 2016. ICANN held its 18th AGM and the public meeting (ICANN57) in Hyderabad. Alongside the public meetings, NASSCOM CoE-IoT organized a hackathon on Smart Cities at T-Hub, Hyderabad. The hackathon was supported by ICANN where the jury included board members along with local academics and business leaders. The hackathon was a big success - with 25 teams working day-and-night on their ideas for smart city solutions. This initial partnership laid the foundation for a more robust association between the two entities.

In June 2018 a Memorandum of Understanding was executed between NASSCOM COE-IoT and ICANN. The MOU supports collaboration on research and engagement activities that contribute to the innovation in Internet identifier technologies and participation in ICANN activities by the wider community in India.

The first project sponsored by ICANN within the scope of this MOU was to experiment with various technologies to explore automated firmware updates for IOT devices in an IPv6 networking context. This project was based on a paper authored by Alain Durand of ICANN and published by IETF (Draft Durand Object Exchange) in December 2017.

SCOPE OF ENGAGEMENT

To set up a testbed and experiment with various technologies to explore automated firmware update strategies in an IPv6 networking context. The team worked with IPv6-only devices, DNS over IPv6, DHCPv6, with the objective to explore how the methodology will fare in the IPv6 only world while facing with real-life network constraints.

The engagement was envisaged to be of 12-month duration in this phase. Depending upon the outcome, future engagements may be planned.

To drive the work, CoE team partnered with Indian Institute of Technology - Hyderabad's (IIT-H) Computer Science department. Annexure A to this document is the final report of the project that was carried out by the students of IIT-H and supported by NASSCOM COE-IoT, its partner ERNET with guidance from ICANN.

We give below the project plan that we followed to execute the project.

PROJECT PLAN

No.	Pillar	Activity	Responsibilities
1	Research	Establish Research Management Team (RMT)	- NASSCOM appoints Point of Contact(s) Sudhanshu Mittal, Director – Industry 4.0 - ICANN appoints Point of Contact(s): Samiran Gupta, Head of India; Matt Larson, VP Research, Office of CTO and Alain Durand, Principal Technologist, Office of CTO
2	Research	Appoint Research Project Manager to lead Research Team	- RMT appointed Praveen Misra, ERNET as technical expert. Praveen Misra worked with RMT to establish timeline and resource plan. Project Management was handled by NASSCOM team initially consisting of Pulkeshian Daruka an later on Sanjay Kumar Mittal, NASSCOM. Sudhanshu Mittal kept the complete oversight of the project.
//3	Research	Establish Research Team (RT)	- Praveen Misra and Sudhanshu Mittal interviewed various universities and eventually engaged IIT-H to undertake the project. The IIT-H team included: • Dr. Bheemarjuna Reddy Tamma, Associate Professor • Dr. Antony Franklin, Associate Professor • Tulika Agarwal, M. Tech student • Madhura A, M. Tech student - Research Team embarked on research according to established timeline and resource plan. - Research Team set up calls initially on a weekly basis and then in decreasing frequency to update RMT as agreed. - Facilities required were provided by NASSCOM or at partnered university. - The Research Team setup testbed in their Computer Science Laboratory.

4.	Related Activities	Visit by David Conrad, CTO, ICANN Visit by Alain Durand, Principal Technologist, ICANN	Leading up to the signing of the MOU, David visited the CoE Gurugram on 6th June 2018. The purpose of the visit was to discuss the COE's activities with Sanjeev Malhotra and Sudhanshu Mittal and also to meet some of the incubated startups at the CoE and understand their areas of work. Alain visited CoE Gurugram on 11th February 2019 where he addressed the roundtable on challenges and benefit of IPv6 adoption. Participants at roundtable included representation from ARICENT, DLINK, CDOT, CDAC and IIT Delhi apart from representation from ERNET, ICANN and CoE. Shri Rahul Gosain, director at the Ministry of Electronics and IT also participated. Subsequently Alain along with Samiran, Sudhanshu and Praveen visited IIT Hyderabad where the project was discussed in detail with the research team. During the visit Alain also addressed the IIT students to share his knowledge about the role played by ICANN in Internet governance and the necessity of moving to IPv6.

Roundtable at CoE Gurugram

Meeting with Research team at IIT Hyderabad

CONCLUSION:

The successful research engagement has also built up the understanding for the RMT as to how to create the research framework and subsequently engage with academic team doing the actual work. Going forward this knowledge can be used to drive more effective research activity.

As CoE has discussed this research engagement with different stakeholders in government and Industry, there is keen interest in the future research engagements and that has led to creation of pipeline as mentioned above. Presently the cyber security is a big area of interest for the CoE stakeholder. While both the engagements listed above have security as a key aspect, the DNS Security is recommended as the area where CoE should focus for next research engagement. To take this forward, the CoE team will come up with a concept note and potential list of academic partners who can be assessed to drive the research activity and share with the ICANN team.

ANNEXURE A – Detailed Project Report

The final report submitted by the research team at IIT Hyderabad is below.

IPv6 based IoT device Firmware Update using DNS: A Performance Study

Dr. Bheemarjuna Reddy Tamma
Associate Professor
IIT Hyderabad
tbr@iith.ac.in

Dr. Antony Franklin
Associate Professor
IIT Hyderabad
antony.franklin@iith.ac.in

Madhura A
M.Tech
IIT Hyderabad
cs17mtech11013@iith.ac.in

Tulika Agrawal
M.Tech
IIT Hyderabad
cs17mtech11022@iith.ac.in

Acknowledgements

We wish to thank Mr. Alain Durand (ICANN), Mr. Samiran Gupta (ICANN India), Mr. Sudhanshu Mittal (NASSCOM) and Mr. Praveen Mishra (ERNET India) for their valuable time, technical support and continuous encouragement in this project. This project work was funded by ICANN through NASSCOM Center of Excellence for IoT.

Table of Contents

I. Abstract

IoT based product lines have seen a vast amount of activity over the previous decade. This activity is anticipated to expand over the years with an estimated projection of billions of connected IoT devices. One of the major concerns with IoT is making sure the devices are secure as majority of them are equipped with limited resources. IoT devices also have the possibility of missing some critical firmware updates, especially when they are on-the-shelves for a long period of time before getting deployed in the field. This leads to the devices being highly vulnerable until the devices get patched with the latest firmware updates and bug fixes from IoT manufacturers. It is therefore essential to provide IoT devices with firmware updates in a timely manner. Towards this, this work relies on a unique Resource Record (RR) called Object Exchange (OX) which is defined in IETF draft <**draft-durand-object-exchange-00**> for the exchange of digital objects using identifiers stored within the DNS, and proposes various firmware update logic mechanisms for automatically updating the firmware of IoT devices over the internet. Evaluations of the proposed updation logic mechanisms have been performed on an exclusive IPv6 based IoT environment. The robustness and scalability of the proposed firmware update logic mechanisms using OX RR over DNS infrastructure have been studied extensively by setting up a testbed of 3,000 emulated IoT devices under varying network conditions. The experimental results show that the name server of the IoT manufacturer containing firmware version details in OX RRs is able to serve the firmware updation requests from 3000 devices like the way it handles requests from one device. Our observations of packet loss reveal that even though we have packet loss of 75% in the network, the update logic works with a success rate of 82% in patching the devices to the latest firmware version.

II. Introduction

Internet of Things (IoT) is a network of devices which sense, accumulate and transfer data over the Internet without any human intervention. The basic nature of IoT device is to collect data from its surrounding environment, process it, and send it to a server or central repository over the Internet. IoT includes devices of all shapes and sizes ranging from low-cost sensors like NodeMCU which could be configured to report environmental readings or wearable fitness devices that measure heart rate to high-end smart microwaves which automatically cook the food, or self-driving cars which detect objects coming in their path using complex sensors and computationally heavy algorithms. According to Juniper [1], the number of IoT devices is expected to grow by 150%, from an estimated 21B in 2018 to 50B in 2022 and the global traffic will grow even more than sevenfold over this period. This is because of massive increase in deployment of video applications, increase in the use of applications such as smart car navigation systems which require greater bandwidth and lower latency, the rapid development of low-power electronics and data analytic techniques.

With this huge number of IoT devices making their foray into the connected world, their deployment in uncontrolled, complex, and hostile environments brings in many new issues as elucidated below.

- One of these issues is the migration of IoT devices onto IPv6 networks. Cisco VNI forecast [2] states that IPv4 address space is already exhausted and IPv6 protocol stack offers additional advantages. However, transitioning from IPv4 to IPv6 network is a tedious and costly exercise. NAT mechanism is helping to cleverly circumvent the problem of limited IPv4 address space.

- IoT devices generally come with factory-installed firmware. These devices need to be updated as soon as new patches are released, without which a large portion of the network might get compromised. Currently, each manufacturer has its own proprietary method of performing firmware updates, and there is no open standard to deal with this issue. In fact, in the case of URL breakage of manufacturer's firmware repository, for example due to acquisitions or mergers, it is very challenging to update the firmware of IoT devices that are already deployed in the field.

- Device's firmware updation is based on either push or pull-based protocols. In pull-based system, IP addresses of the firmware repositories should be embedded into the IoT devices. If the IP address of the firmware repository changes at a later point of time, the IoT device may not get updated with the new IP address of the repository. While in a push based system, the reachability information of the IoT device must be updated at the central repository. Hence, both the pull-based system (using embedded IP addresses) and push-based system are not appropriate choices to update the firmware of IoT devices in a reliable manner.

These issues make timely firmware updation of IoT devices quite challenging, failing to apply patches in a timely manner can lead to hacking risks and even loss of lives. For example, almost half a million pacemakers were recalled by the US administration [3] due to the fear of an issue in a firmware update which could lead to loss of millions of lives. Another example is Silex malware [4] which wiped out the firmware of IoT devices and halted the device completely. This tells us that the security of devices is not a static process, and one should be constantly vigilant against unauthorized access.

The main contributions are as follows:

1. Creation of IPv6 only network setup containing authoritative name server, emulated IoT clients, firmware server and nsupdate client for studying effectiveness of IoT firmware updation using DNS. All the entities are given global unicast IPv6 address using DHCP

and IPv4 is completely disabled in all of the network entities in the setup.

2. Proposed Update Logics considering various scenarios:
    a. Update logic #1: It uses Model ID of IoT device and version number of the current firmware to get new firmware details from the authoritative name server periodically.
    b. Update logic #2: It considers the security issue that can be raised by sending Model ID of IoT device in the DNS query.
    c. Update logic #3: It takes care of critical and non-critical types of firmware updates. It helps in saving energy of IoT devices by delaying the firmware updation process in case of non-critical updates.
    d. Update logic #4: It defines IoT client side behavior when the client is running multipart firmware.
3. We used DNSPython library for implementing the proposed update logics. In DNSPython, there was no support for OX RRs. Hence, we modified DNSPython library to add OX RR support.
4. In recent BIND releases (from 9.12.3), DoA RR support is available, but not for OX RR. Hence, we modified **BIND** software to rename DOA RR to OX RR.
5. Emulation of 3000 IoT devices using docker-compose to conduct various experiments for performance testing. This is done using 10 machines, by running 300 dockers on each machine in parallel.
6. Demonstration of secure update of OX RRs at the name server using **nsupdate** to check the effect on response while nsupdate is updating the DNS**.**
7. Conducted experiments in multiple failure cases using up to 3000 emulated devices in parallel to see the performance.

The rest of the report is organized as follows -

1. In Section III, we present the background of the DNS and how is it used with OX record.
2. The challenges in the process of firmware updation of IoT devices faced by the current update procedures have been described in Section IV.
3. A detailed description of the roles of different components along with a block diagram of how to use DNS OX record for firmware update has been shown in section V.
4. Section VI presents the overview of the update logics being used in our work and different scenarios to update the firmware.
5. Section VII shows how manufacturers can update the contents of DNS zone file.
6. Section VIII shows the experimental testbed setup being used for simulation scenarios and the performance evaluation of Authoritative name server.
7. Finally, we conclude the report in Section IX.

III. Background

1. Domain Name System (DNS)

The Domain Name System (DNS) is a central part of the Internet. It acts as a domain directory by providing bidirectional translation between domain names and IP addresses using a globally distributed database [6]. The distributed database of the DNS infrastructure is hierarchically organized having an inverted tree-like structure with **Root Name Servers** managing the root of the hierarchy. To have a decentralized structure, there are 13 root name servers that are globally distributed. Under the root node, there are multiple Top Level Name Servers which include name server for Country Code Top Level Domains (ccTLDs), generic Top Level Domains (gTLDs) and unsponsored TLDs. The top level name servers are followed by one or more Authoritative name servers in the path, which manage the information about subdomains in the name space.

The process of translating the domain name to the IP address is known as **DNS Resolution**. To resolve a domain name say "iith.ipv6.ernet.in", the client forms a **DNS query** and initiates the DNS resolution process by sending out a query to the local resolver, which in turn contacts one of the root name servers. The root name server returns the IP address of the authoritative name server for the domain ".in". The resolver then sends the query to the ".in" authoritative name server to find an authoritative name server for "ernet.in", which will be queried again for ipv6.ernet.in. The process continues iteratively until the IP address of iith.ipv6.ernet.in is retrieved from an authoritative name server for that domain.

The name servers in the DNS infrastructure store the information about the domains in a simple text file called **Zone files**. Information about the domains in the zone file is encoded and stored using different types of **Resource Records (RR)**. The most common RR types include A RR which stores domain name to IPv4 address mapping, AAAA RR which stores domain name to IPv6 address mapping, CNAME RR which identifies alias names for domain names, NS RR which identifies the authoritative name server, and MX RR for specifying the mail server responsible for accepting email messages on behalf of a domain name.

Basically DNS was originally designed for naming mail servers, and subsequently it became the standard for mapping IP addresses and domain names. However, its success as a lightweight translation mechanism has led to the expansion of DNS infrastructure for other related applications like host integrity identification [7], replica selection for content delivery networks [8], certification authority information [9], the neighbor discovery process in IPv6 [10], customer management information [11], etc.

While most DNS transactions occur over UDP, DNS by design can utilize either TCP or UDP as the underlying transport protocol. The maximum allowable DNS response size using UDP is 512

bytes. When the response size is beyond 512 bytes, TCP is used. Due to an increase in the deployment of IPv6 networks and extensions to DNS, the use of TCP for DNS messages is also increased in the recent past. As the functionality of the DNS protocol is getting increased, the size restrictions of the DNS response led to the introduction of Extension to DNS (EDNS) [12].

2. Using DNS for Object Exchange

In addition to hostname to IP address mapping, DNS can be used in many internet-related applications and one such application of DNS is the exchange of digital objects using identifiers stored within the DNS using Object Exchange (OX) resource record (RR) defined in IETF draft [13]. Each OX RR contains an object having various fields that could be private to the producer and the consumer. An OX RR can either hold the data directly or gives a pointer to the location where data is stored.

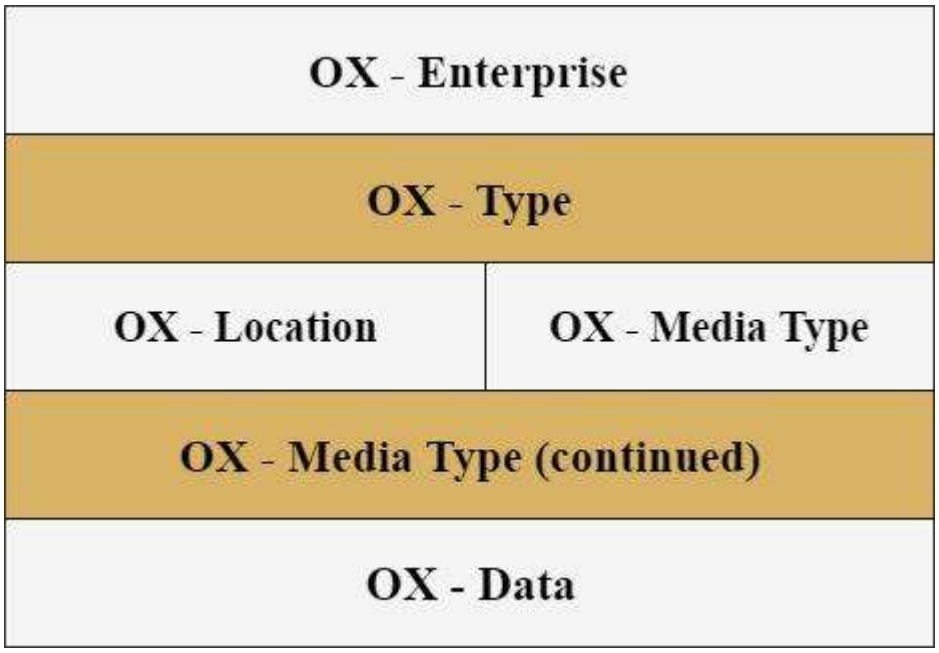

Fig. 1: OX Resource Record Format

Fig. 1 shows the format of OX RR where five fields are defined: OX-ENTERPRISE, OX-TYPE, OX-LOCATION, OX-MEDIA-TYPE, and OX-DATA. The combination of OX-ENTERPRISE and OX-TYPE fields is used to indicate the semantic type of the OX RR.The interpretation of the OX-DATA fields is governed by OX-LOCATION field. According to IETF draft [13], interpretation of OX-LOCATION and OX-DATA field is as follows.

1. OX-LOCATION value is 1: OX-DATA contains actual OX object
2. OX-LOCATION value is 2: OX-DATA stores UTF-8 encoded string indicating the URI from which the object can be obtained.

3. OX-LOCATION value is 3: OX-DATA holds UTF-8 encoded string representing the handle from the Handle System [14] from which the OX object can be obtained.

The OX-MEDIA-TYPE field contains the Internet media type [15] for the OX object represented by this record.

Now-a-days, IoT devices, with full IP stack, send sensor information to remote servers over the Internet. These devices all the time rely on DNS infrastructure to get to know the IP addresses of these remote servers. So, it will be a good idea to make use of DNS infrastructure by some means for firmware update of IoT devices.

A prototype of updating IoT firmware using DNS infrastructure was presented in the ICANN60 meeting 2017 [5], by **Alain Durand, ICANN** and **Fernando Lopez, National University of La Plata**. In general, the URL breakage can happen due to many reasons such as - organizational changes, mergers, acquisitions, company name changes, etc. There are various existing solutions, like URL redirection, URL shortening, etc. However, these solutions do not provide persistent identifiers. This can be achieved using the new DNS RR, Object Exchange (OX). To demonstrate the firmware update of IoT devices using OX records [27], a few NodeMCU devices were utilized in the prototype system [5]. In the experiment, which was setup on IPv4 network, an IoT device sends a DNS query for the OX record to the name server of IoT manufacturer and receives firmware URL in the form of a DNS response, which is used to download and update the firmware of IoT device. The device model number and enterprise number were used to match the DNS OX query in the authoritative name server before sending the response. In that effort, the ESP8266 LWIP library [28] of NodeMCU was patched to support OX records. A small Django 1.11.6 application was also developed for updating DNS zone files by performing CRUD operations.

IV. Challenges in updating IoT devices' firmware

As the number of IoT devices connected to the Internet are growing exponentially, more and more firmware vulnerabilities are being discovered, that have raised the need for a simple, standardized, secure and timely firmware update mechanism. One mechanism available which helps to keep the IoT devices secure and up-to-date is *Firmware Over The Air (FOTA)*, which refers to the mechanism of remotely updating the code of an IoT device. Based on the mode of operation, there are 3 different kinds of FOTA methods available as discussed in [16].

1. Server-initiated
2. Client-initiated
3. Hybrid mode

Server-initiated FOTA is also known as push-based firmware update. In this method, there is a status tracker that identifies the list of IoT devices which are eligible for a firmware update. For

the status tracker to select a device, the reachability information of the IoT device must be up-to-date at the status tracker. In client-initiated FOTA method, IoT devices proactively check for firmware updates on the firmware server and pull the images if they are any latest ones. In the hybrid mode of operation, the status tracker notifies the availability of the firmware to the IoT devices and the IoT devices also can pull the image as soon as it is possible.

One of the main challenges in server-initiated FOTA procedure is in getting the reachability information of the IoT devices at the status tracker, because it could be difficult to obtain under adverse network conditions. Also, IoT devices that are brought in, but not connected to any network will lose many updates or security patches, in the worst case they may not be updated at all.

As compared to server-initiated FOTA procedure, client-initiated or pull-based FOTA procedures are not well studied in the literature. One example of a client-initiated FOTA procedure is in IETF draft [13], which uses DNS infrastructure for the firmware update. The idea here is that IoT devices proactively query the name server of the firmware manufacturer (OEM) using 'OX' resource record and the name server will then send the response containing the information about where and how the IoT device can get the firmware updates. Although this strategy appears to be correct, this has not been extensively researched in the literature. So the main focus of this report is on reusing current DNS infrastructure for FOTA by coming up with light-weight client-initiated FOTA mechanisms with minimal functionalities and evaluating their performance.

V. IoT Firmware Updation using DNS

In this method, DNS OX resource records are used to update the firmware of IoT devices. The OX queries are standard DNS queries similar to A and AAAA queries. The OX record fields can be used as follows in the firmware update process:

1. The OX-Type field is used to tell the kind of data stored in the Data field, such as the Firmware URL, the firmware hash-value, the firmware version number, etc.
2. The OX-Location field is used to indicate whether the information is stored locally on the name server or the information given is URI.
3. We did not use the Media-Type field in the Firmware Update process.

The field values and their use are entirely private to the manufacturer (OEM).

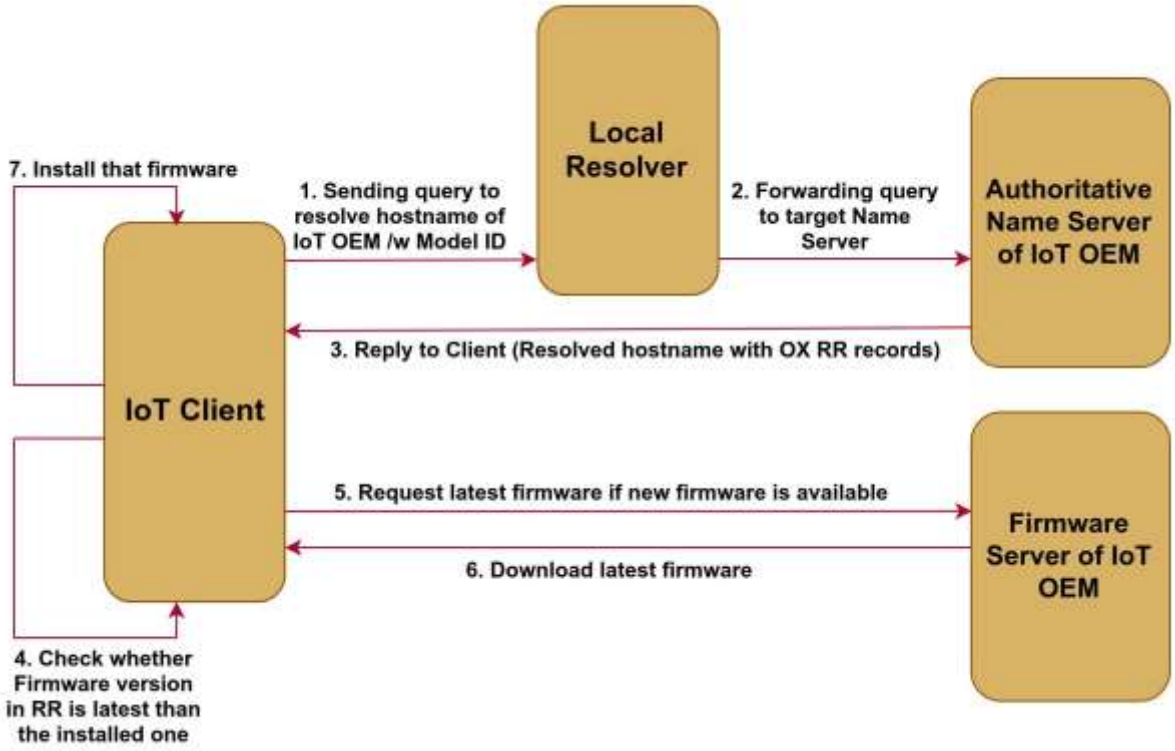

Fig. 2: Firmware Update System Model

An illustration for the firmware update process is shown in Fig. 2. The basic idea is that IoT client will generate a query for an OX record to verify if the firmware running on it is the latest or not. The Authoritative name server of the OEM stores the firmware URI and its version number in the OX records. In Step-1, the client will first send an OX query to the authoritative name server requesting the OX resource records. Here, the Local Resolver will follow the whole DNS hierarchy to reach the Authoritative name server. After getting the address of Authoritative name server, the resolver will send that query to the name server. The Authoritative name server matches that OX query with the stored resource records, and responds accordingly. It will send the firmware URI and its version number to verify if the device needs to update the firmware or not. After getting the response from the server-side in Step-3, the IoT client compares the firmware version presently operating on it with the firmware version it received in the OX response in Step-4. If the IoT device is not running on the latest firmware, it contacts the firmware server using the firmware URI received in the OX-Response in Step-5. Firmware server of the manufacturer stores all the firmwares available for all the device models. The device downloads the latest firmware image in Step-6, and installs it finally in Step-7.

VI. Update Logics

An update logic enables an IoT device to verify the running firmware and offers a way to update the firmware safely in case of a new firmware availability. The client device checks for firmware updates periodically or whenever it reboots. In this work, we have proposed a few Update Logics, each of them is described in detail in the following.

a. Update Logic #1

This update logic uses the Model ID of IoT devices. Here, the assumption is that the client always knows its Model ID and the current firmware version running on it. To match the incoming query, the Authoritative Name Server stores the Model ID and firmware version number in OX records.

Client Query Format:

Fig. 3: Client Query Format for Update Logic #1

Name Server RR Format:

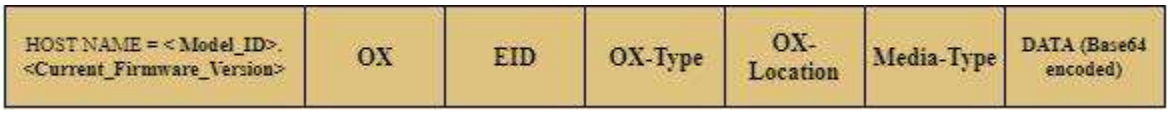

Fig. 4: Server Record Format for Update Logic #1

In the OX query, the IoT client sends its Model ID and the current firmware version as shown in Fig. 3. The OX RRs at the name server have the format as described in Fig. 4. When the OX request is received, the name server matches the records stored in it using Model_ID.VersionNo and sends the firmware URI of the next compatible version and the firmware version number. Table 1 describes various OX-type values used in this update logic along with their field specifications.

Table 1. OX Field Specifications of Update Logic #1

OX-Type	Purpose
101	Firmware URI of next compatible version
102	Version number of next compatible version

Client Side Process:

Step 1. Query = < Model_ID >.< Current_Firmware_Version >.< Domain_Name >
Step 2. Response = Response to above query (OX records)
Step 3. Iterate over the received resource records:
If rr.type == 102
New_version = rr.data
If rr.type == 101
Firmware URL = rr.data
Step 4. If current firmware version < New_version
Fetch new firmware and install it
else
Goto Step 1 after a time interval

There are two scenarios to update the IoT device's firmware. The first scenario is **Direct Update**, where there is no such dependency in which the device can update directly from the running version to the latest available version. Another update process situation is **Sequential Update** in which a device cannot update straight to the recent version due to some dependencies on previous versions. Consider the Table 2 in which an example of various firmware versions and their dependencies are given for a device with Model ID=9811.

Table 2. Example of OX records at Name Server

Domain Name	RR	EID	Type	Location	Media type	Data (Base64 encoded)
9811.1	OX	12	101	2	Text/plain	URI of V2
9811.1	OX	12	102	1	Text/plain	2 (Version)

9811.2	OX	12	101	2	Text/plain	URL of V3
9811.2	OX	12	102	1	Text/plain	3 (version)
9811.3	OX	12	101	2	Text/plain	URL of V5
9811.3	OX	12	102	1	Text/plain	5 (version)
9811.4	OX	12	101	2	Text/plain	URL of V5
9811.4	OX	12	102	1	Text/plain	5 (version)
9811.5	OX	12	102	2	Text/plain	5 (version)

1. **Sequential Update**

In the example given in Fig. 5, the client is running on firmware version 1. It sends an OX query to check for available firmware updates. According to the given case in Table 2, the name server will send an RR with URI of firmware version 2 and another RR specifying the version as 2. The client will compare its current version and the version it received from the name server. Since the received firmware version number is greater, the IoT device will fetch the firmware from the specified URI and installs it. The client sends OX query again after reboot to check whether it is running the latest version or not. This time, the client will get the firmware version 3. It will install that with the same process described above and reboots. Again, to get the confirmation, the client will send the OX query and receives URI of version 5. After installing version 5, it again sends out OX query to check whether any other firmware updates are available. But, this time the client will get the URI of version 5, which is the same as running firmware, so it will stop the update process. Since the device has to check periodically for the firmware update, it will again send an OX query to the name server after the specified time interval.

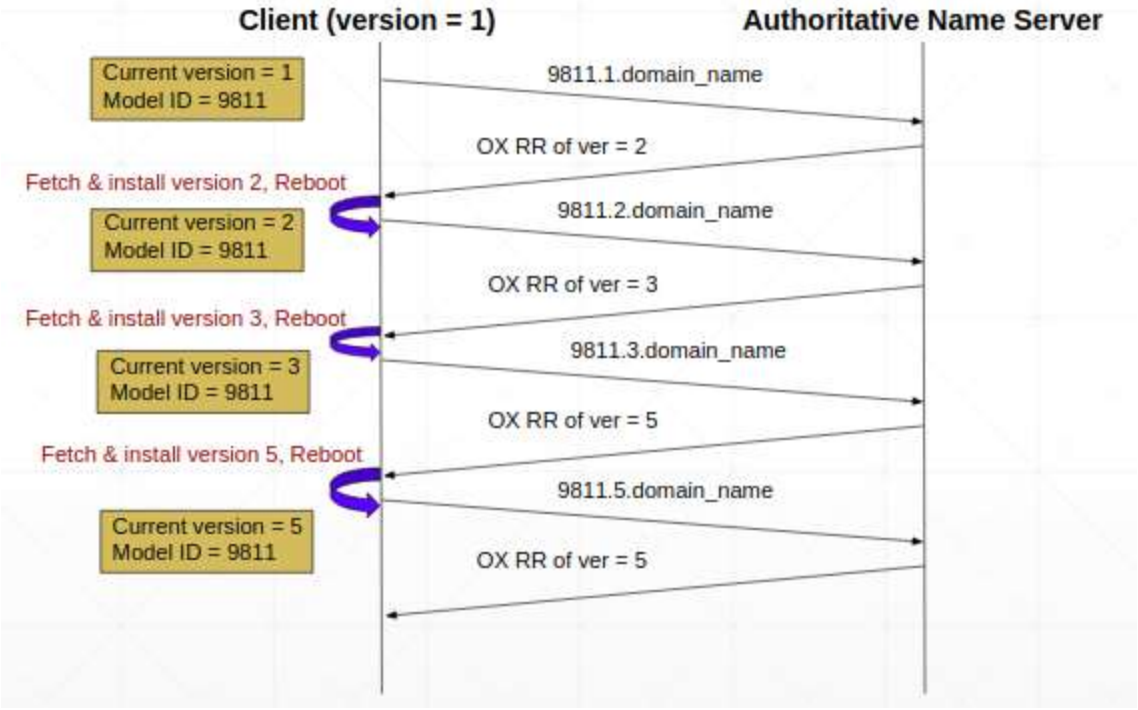

Fig. 5: Sequential Update Scenario from Version 1 to Version 5

2. Direct Update

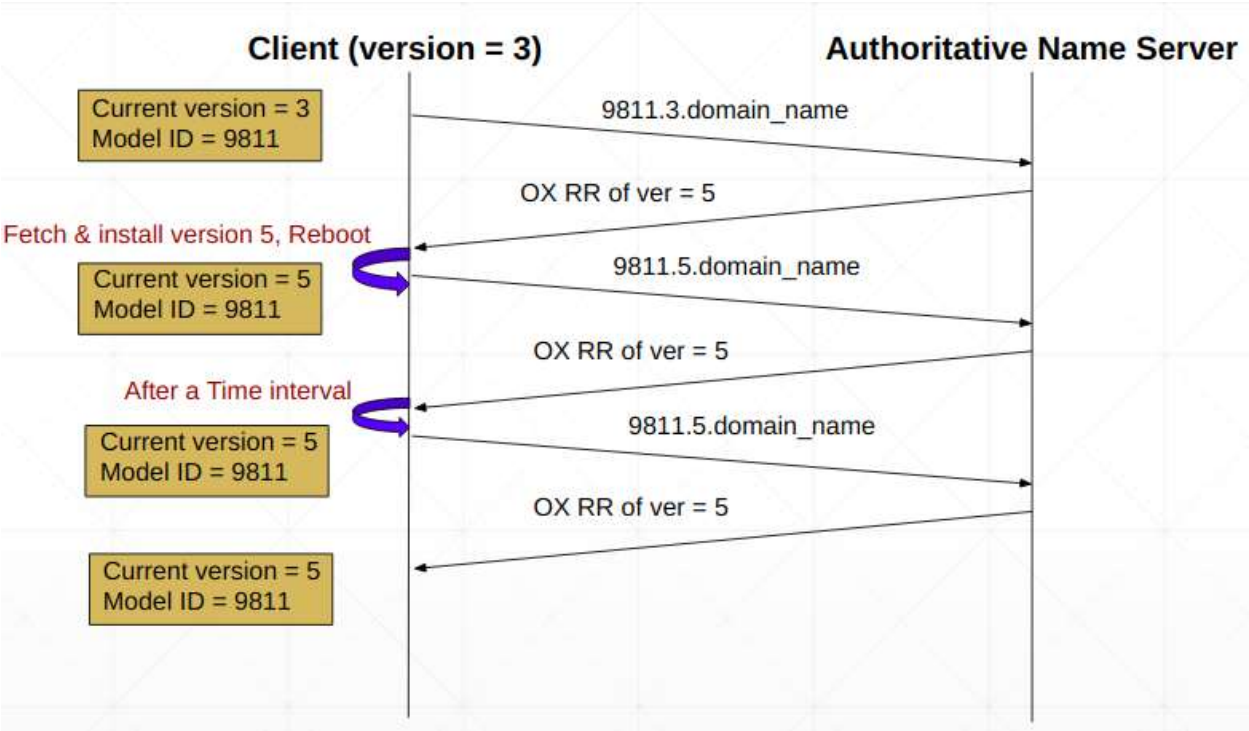

Fig. 6: Direct Update Scenario from Version 3 to Version 5

In the example given in Fig. 6, the client is running on firmware version 3. It sends an OX

query asking for any firmware updates. According to Table 2, the name server will send the URI of firmware version 5 and an RR mentioning the version in URI is version 5. Since the received firmware is the latest, the IoT device will fetch the firmware from the specified URI. It will check the firmware received and install it if there is no problem found. Again, to get the confirmation, the client will send out an OX query. This time the device will get the URI of version 5, which is the same as the running firmware, so it will stop the update process.

Pros: The benefit of using Update Logic #1 is that on the server-side, there is no need for extra computation. Besides this, the size of the DNS response is also the least possible, as it only contains Firmware URI and its version number.

Issues: Querying for the OX record using Model ID and current version running on IoT device has certain disadvantages in terms of security of these devices. If there is an attacker who gets to know about the current version running on the IoT device, then the attacker can launch Man-in-The-Middle (MITM) attack to perform malicious activity by sending corrupted firmware which will be having the compatible version to install.

b. Update Logic #2

To overcome the disadvantage of Update Logic #1, instead of sending Model ID and the current version, Hash-values will be used in the proposed Update Logic #2. This work uses SHA-2 (224 bits) technique for obtaining hash values of the firmware image. SHA-224 is utilised as the domain name cannot contain more than 63 characters between the two dots. On the name server side, it uses these hash values to match the query and then responds accordingly. Since images of different firmware versions will have different hash-values, there is no need to explicitly include the firmware version here in the OX response. The format in which client queries is shown in Fig. 7. Fig. 8 describes the OX RR format stored in the name server.

Client Query Format:

Fig. 7: Client Query Format for Update Logic 2

Server RR Format:

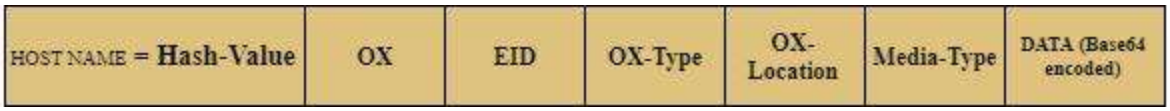

Fig. 8: Server Record Format for Update Logic 2

Table 3. OX Field Specifications for Update Logic #2

OX-Type	Purpose
101	Firmware URI of next compatible version
102	Hash-value of next compatible firmware version

The above Table 3 represents the field specifications used for update logic #2. These values and their mapping entirely depend upon the manufacturer. Firmware URI will store the next compatible version of the firmware URL that needs to install on the IoT device. Hash-value of firmware will ensure the security of IoT devices against MITM attacks. The device will compute the hash-value of the firmware received, and it will match that value with the hash value of the firmware received in OX response to verify whether the firmware is compromised or not.

We do not need to store the firmware version separately as we can directly compare the firmware using their hash values.

Client Side Process:

Step 1. Query = < Hash_Value_Current_Firmware >.< Domain_Name >
Step 2. Response = Response to above query (OX records)
Step 3. Iterate over the received resource records:
If rr.type == 102
Hash_Value_New = rr.data
If rr.type == 101
Firmware URL = rr.data
Step 4. If Hash_Value_Current_Firmware != Hash_Value_New
Fetch new firmware and install it
else
Goto Step 1 after a time interval

Handling various issues during update process in Update Logic #2:

1. **Handling Packet Loss/ Delayed Response:** In case of packet loss or delay in getting OX responses, these scenarios can be detected using timeouts. There will be a specified timeout

at the client side for the response to reach the client. If the IoT client does not get a response within that period, there will be an exception of Delayed response.

In DNS, there is a lifetime period specified. Lifetime period of a DNS server specifies the time until which a query can be retried in case of a timeout. In cases of huge packet loss or non-availability of the name server, the client might not get the server's response within the lifetime period. Since firmware updates are usually queried at fixed intervals, this situation leaves the client vulnerable until the next retry. When the name server is unavailable, repeated retries from multiple clients will drastically increase the server load. To mitigate this issue, a random exponential backoff algorithm is used to assign a randomly generated waiting time to each client in case of failure scenarios. The waiting time for next OX query will increase exponentially with each packet loss or delayed response.

2. **Client runs Corrupted Firmware:** The IoT device sends an OX query by generating the hash-value of its firmware to get the firmware update. In case of corrupted firmware, the name server will not be able to match that hash-value with any of the stored OX records. The name server will throw an exception of 'NX-Domain Error'. To mitigate this issue, the name server maintains default entries for each Model ID of the IoT devices. These default entries have the URI of the firmware that can directly be installed on the given Model ID without any dependency issues. When the client gets this exception, it will immediately send an OX query for the default entry using its Model ID.

As specified in the update logic #1, in update logic #2 also, we have direct and sequential update scenarios. Table 4 shows an example of OX RRs stored in the name server in case of update logic #2.

Table 4. Example of OX records at Name Server

Domain Name	RR	EID	OX-Type	OX-Location	OX-Data
h1	OX	12	101	2	URI of V2
h1	OX	12	102	1	h2
h2	OX	12	101	2	URI of V3
h2	OX	12	102	1	h3
h3	OX	12	101	2	URI of V5
h3	OX	12	102	1	h5
h5	OX	12	101	2	URI of V5
h5	OX	12	102	1	h5

ModelID (default)	OX	12	101	2	URI of compatible firmware
ModelID	OX	12	102	1	hc

1. **Sequential Update**

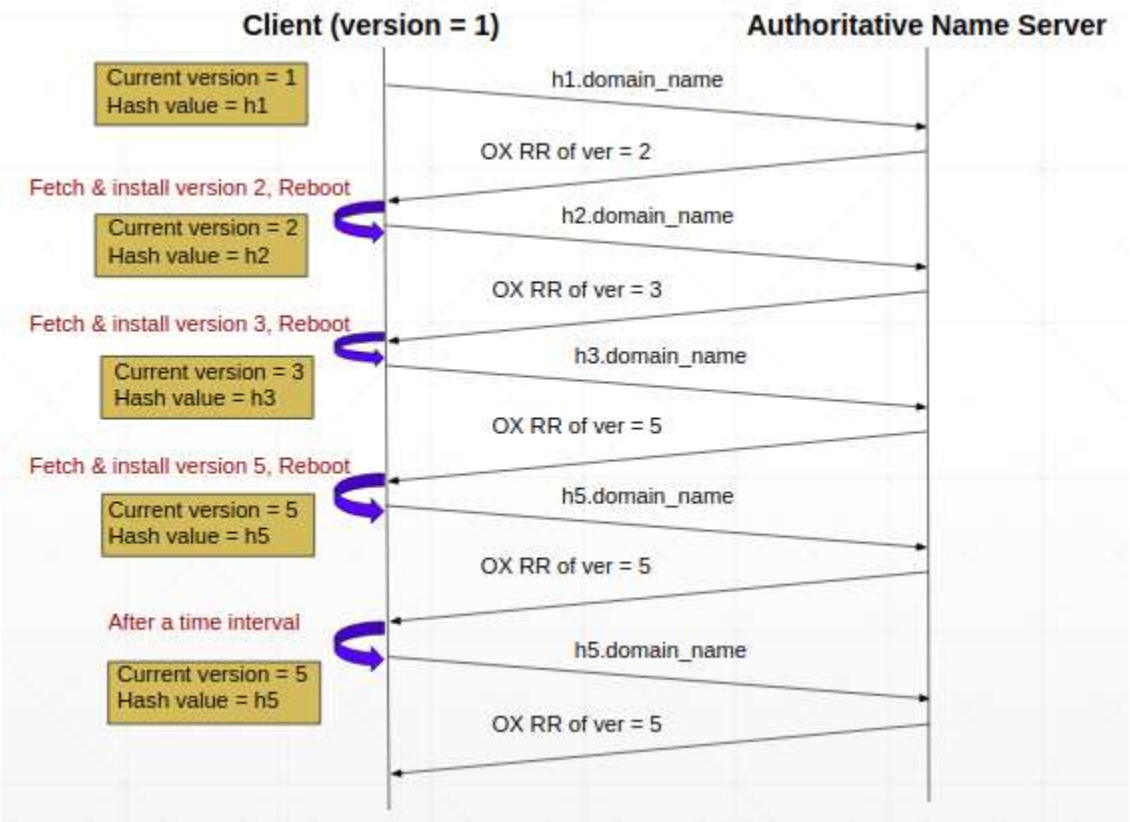

Fig. 9: Sequential Update Scenario from Version 1 to Version 5

In the example given in Fig. 9, the client is running on firmware version 1. It sends an OX query to check for firmware updates. According to the given case in Table 4, the name server will send the URI of firmware version 2 and its hash value. The client will compare its current hash-value with the hash value received in OX response. If they do not match, it means that the received firmware version is the latest, so the IoT device will fetch the firmware image from the specified URI. It will calculate the hash-value of the firmware downloaded from the URI and will compare it with the received hash-value in OX response. If the hash-values do not match, the client will get to know that the received firmware is corrupt and it will not install else it will install the received firmware. The client sends the OX query again to check whether it is running the latest version or not. This time, the client will get the firmware version 3. It will install that with the same process described above. Again, to get the confirmation, the client will send the OX query. This

time it will get the URI of version 5. The client will install the firmware version 5 and to get the confirmation about the latest version, client sends the query again. This time it gets OX RRs of version 5, which is the same as running firmware, so it will stop the update process. Since the device has to check periodically for the firmware update, it will again send an OX query to the name server after the specified time interval.

2. **Direct Update**

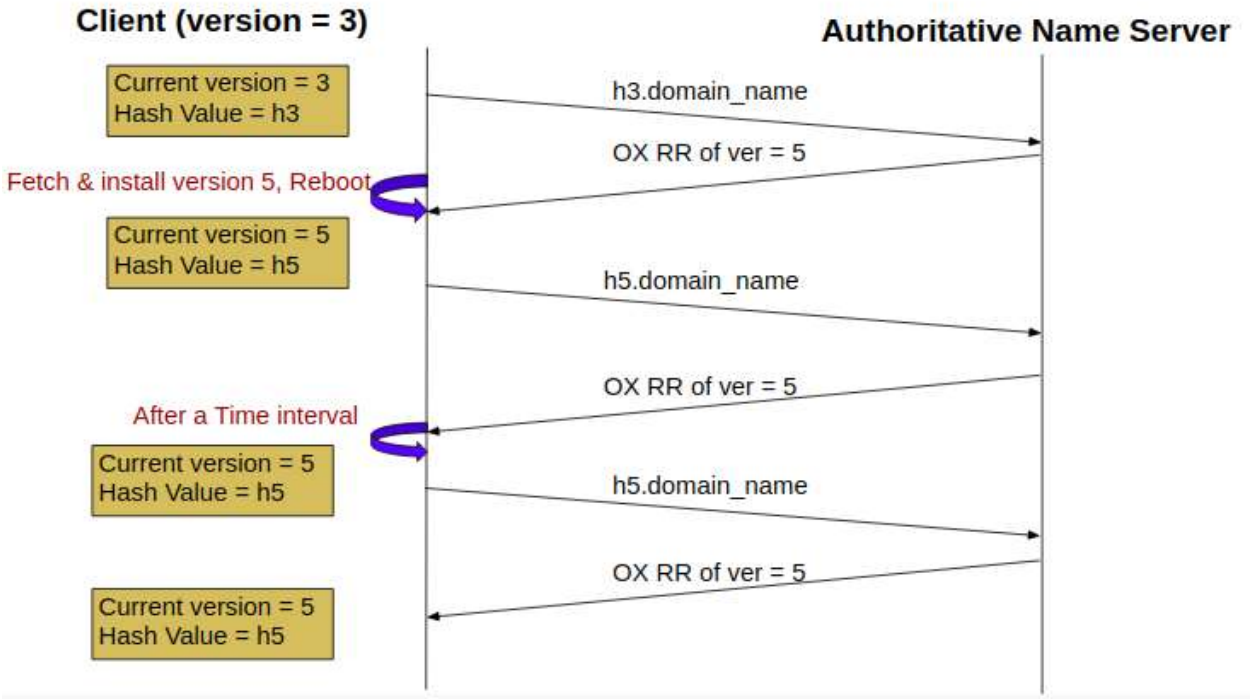

Fig. 10: Direct Update Scenario from Version 3 to Version 5

In the example given in Fig. 10, the client is running on firmware version 3. It sends an OX query asking for the firmware update. According to Table 3, the name server will send the URI of firmware version 5 and its hash value. Since the received firmware is the latest, the IoT device will fetch the firmware from the specified URL. It will check the firmware received and install it if there is no problem found. Again, to get the confirmation, the client will send another OX query. This time the device will get the URI of version 5, which is the same as running firmware, so it will stop the update process. The device will send next query for checking any new firmware update only after a time interval.

c. Update Logic #3:

This logic is defined to handle critical and non critical firmware updates. The manufacturer can define a firmware update as critical, so that IoT clients can install them immediately when they receive as described above in case of Update Logic #1 and #2. For handling

critical and non critical updates, and reduce frequent updates/reboots, we define a cycle called update cycle whose value is few days depending upon how frequently manufacturer releases updates. For defining whether a firmware release is critical or not, we use OX-Type field value as 103 and corresponding OX-Data value as 1 or 0 to indicate critical or non-critical update, respectively. When an IoT client receives a firmware with critical flag, the client coordinates with other processes running in the system and schedules the update immediately. For a non critical update, the client can postpone the update to the beginning of the next update cycle to reduce the number of reboots. The other type field values and the query format for update logic #3 is same as that of update logic #2.

For example, consider a scenario where a manufacturer has two firmware versions: version 1 and version 2, version 2 being the latest and the duration of update cycle is 30 days. Let us suppose the manufacturer has released version 3, version 4 and version 5 within the update cycle which are non critical and released version 2.1 which is being critical. Fig. 11 shows the update process followed by an IoT device which is currently running with the firmware version 2.

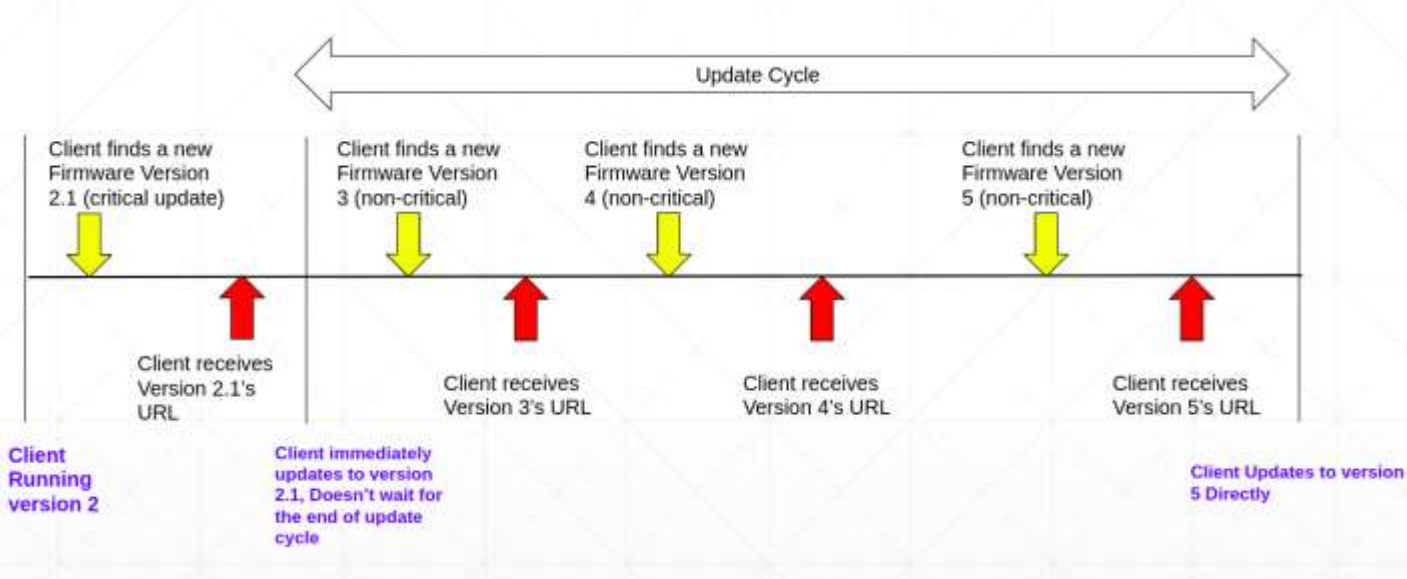

Fig. 11: Critical/ Non Critical Update process in Update Logic #3

As described in Fig. 11, when a client running version 2 finds a new firmware version 2.1 which is critical, the client immediately fetches the firmware version 2.1 from the URL mentioned in the OX RR and installs it. After installing it, a new update cycle of 30 days starts. Within the update cycle, firmware versions 3, 4 and 5 are released and all of them are non critical. Since all are non critical, the client just waits for the update cycle to end. At the beginning of the next update cycle, the client just installs firmware version 5. If the client has some dependencies and cannot directly install the latest version, then the client has to follow the sequential update process shown in Fig. 9.

Pros:

When the IoT device has limited bandwidth and computational resources, in such cases, the number of updates performed by the client can be reduced.

d. Update Logic #4

When a firmware is made up of multiple parts and they have dependencies among them, then the manufacturer can use the update logic defined in this subsection. For defining the dependencies among multiple parts of the firmware, a block of OX-Type values from 104 to 200 are used. For specifying each dependency, the manufacturer has to specify the name of firmware on which current firmware is dependent and the hash value of its minimum required firmware version.

For example, consider a scenario in which the firmware is made up of three parts, firmware A, firmware B, and firmware C. For firmware A to update from version 1 to version 2, firmware B has to be of version 3 and firmware C has to be of version 4. Fig. 12 explains the process taken by a client running firmware A of version 1, firmware B of version 2 and firmware C of version3.

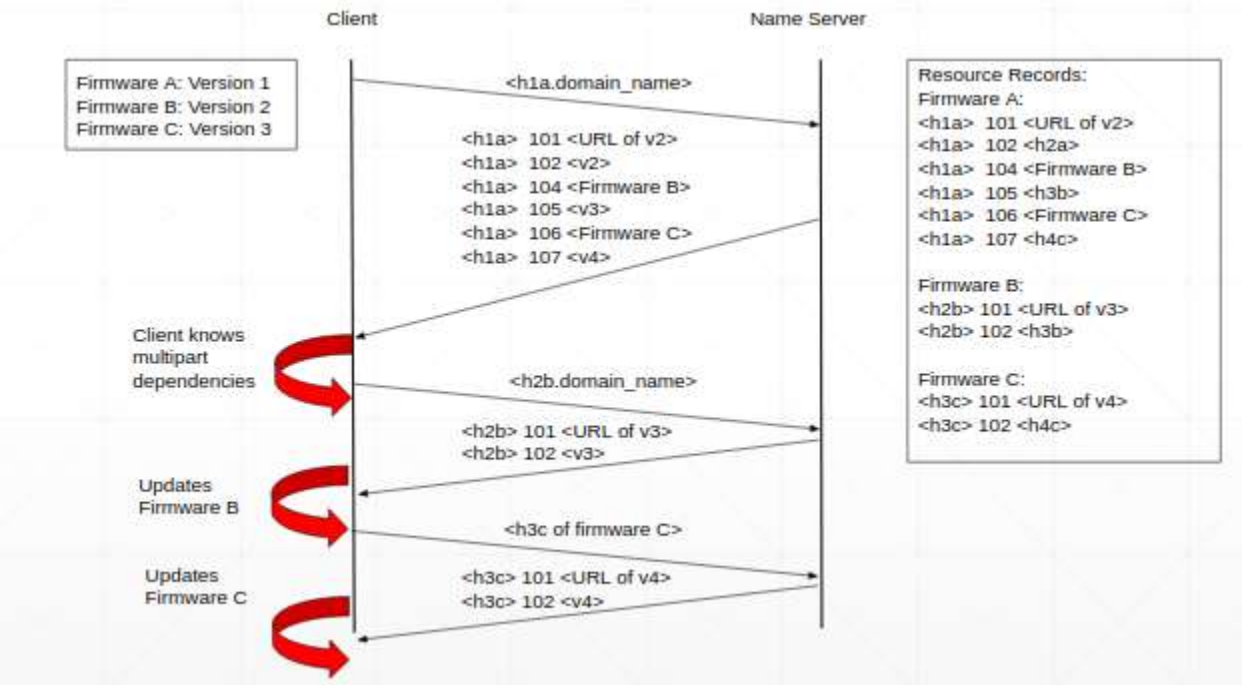

Fig. 12. Multipart Firmware Update Process in Update Logic #4

As described in Fig. 12, a client running version 1 of firmware A queries the name server

of next version of firmware A by sending hash value of version 1 of firmware A (h1a). The name server replies with OX RRs of version 2 of firmware A (h2a) along with other OX RRs with type values 104-107 describing the dependency of version 2 of firmware A with firmware B and firmware C. When the client receives the reply from the name server, it learns about dependencies. It updates the firmware B from version 2 to version 3 by querying with hash value of version 2 (h2b) and gets URL of version 3 of firmware B and hash value of version 3 of firmware B (h3b). It updates firmware B to version 3. Similarly, the client also updates firmware C to version 4 (h4c) before updating firmware A to latest version.

VII. How manufacturer can update DNS Zone file?

In the previous sections, we have mentioned that IoT devices can find the new firmware or patch to the old firmware from the DNS server using OX queries. OX resource records in the authoritative name server store URI of the firmware along with other pieces of information needed for the firmware update. So, whenever the manufacturer releases a new firmware or a patch to the existing firmware, OX RRs in the name server should also be updated to reflect this change. For this entire firmware update procedure to work smoothly, the manufacturer should have some provision to update the content of the DNS server zone files securely and efficiently.

As most of the manufacturers have their own authoritative name servers at different locations, manually changing the content of the zone file is not a solution. Since the content of the zone file is the entry point of the firmware update procedure, security should be the most important concern here. Only authorized entities should have access to update the content of the zone file. Another essential aspect to consider is the availability of the name server for handling OX queries from the client while zone records are getting updated. DNS infrastructure already has an efficient zone update utility called **nsupdate**, which can be reused for updating the OX records also.

nsupdate is a Dynamic DNS (DDNS) utility that can be used to instruct the name server to update the content of the zone file as defined in [17], without manual intervention. Using nsupdate, the manufacturer can add or delete RRs from a zone without manually editing the zone file. All the DDNS requests use transaction signatures which use Transaction Signature (TSIG) RR type described in [18]. The transaction signature depends upon a shared key which is known only to the name server and nsupdate utility. When the name server receives a DDNS request, it first checks whether nsupdate client has authorization or not using transaction signature. If the client is an authorized one, then the name server modifies the content of the zone file to reflect the new changes. The name server maintains atomicity concerning update and query operations. Since nsupdate is an efficient, secure, and remote way of updating the zone files, the manufacturer can also use this facility to update the OX records whenever required.

The nsupdate utility allows updation of RRs that belong to the same zone. Also, if a new product is released, the manufacturer might require a facility to create a new zone which is not possible using nsupdate. Furthermore, if the manufacturer has more than one product for which more than one zone required, it will be cumbersome to maintain multiple nsupdate utilities. In such cases, we recommend creating a web based user interface using frameworks such as Django [19] and Ansible [20], which allows the manufacturer to perform CRUD operations on multiple zones.

VIII. Performance Evaluation of Proposed Update Logics

a. Experimental Setup

To evaluate the performance of update logic (proposed Update Logic #2) in IPv6 only network, we created a testbed containing the following entities:

1. Authoritative name server
2. Firmware server
3. nsupdate client
4. Emulated IoT devices (up to 3000)

Global unicast IPv6 addresses assigned to all the entities present in the testbed. We have used a globally routable domain, "iith.ipv6.ernet.in" for our experiments. The authoritative name server for the domain "iith.ipv6.ernet.in" holds OX RRs containing various pieces of information related to multiple firmware versions available as well as an AAAA RR which includes IPv6 address of the firmware server. The authoritative name server is setup using BIND 9.12.3 [21], which is patched to rename DoA RR to OX RR. The authoritative name server has been deployed on a machine running Ubuntu 16.04 equipped with Intel Xeon CPU E5-2690 v4 (2.60GHz, 56 cores) and 1G NIC. For emulating 3000 devices, we have used 10 machines which include 6 servers (2.60GHz, 56 cores), 3 workstations (2.70GHz, 8 cores) and 1 desktop machine (2.60GHz, 12 cores) each running Ubuntu 16.04. The servers are connected to IPv6 network through a VLAN while rest of the machines are connected through a Wi-Fi access point which is also configured to IPv6. Each device runs 300 docker containers using docker-compose tool, and individual docker containers run update logic written in Python.

The firmware server in the testbed is responsible for storing various firmware versions. For simplicity purpose, the firmware is a simple python program which can be downloaded and executed by the emulated IoT devices easily as the main objective of the study is to test the scalability of DNS infrastructure to handle IoT firmware updates. The nsupdate client is responsible for updating the OX RRs which are present in the authoritative name server of firmware manufacturer. Using a shared key which is known only to nsupdate client and the authoritative name server, the DDNS update procedure is secured as discussed in the previous

section. Emulated IoT devices which are running in the Docker containers having Ubuntu as the underlying host OS run the update logic, get the new firmware, and execute the latest firmware. The update logic is written using Python, using DNS library available in Python called DNSPython [22]. We have also extended the DNSPython library to add OX RR support.

Table 5 lists the software tools used to implement various entities of the testbed.

Table 5. Entities and software used in the implementation

Purpose	Tools Used
Authoritative Name Server/ Resolver	BIND 9.12.3
OS on Server/VM, Raspberry Pi	Ubuntu 16.04, Ubuntu Mate
Firmware Repository	HTTP server using Apache
DNS queries & Performance Evaluation	Dig tool, DNSpython, DNSperf
DDNS Update	nsupdate
Client Simulation	Docker-Compose

b. Configuration Parameters

a. The default ARP table size in the name server was 128. Because of this, when the number of queries increases, the following error is thrown:

8-Sep-2019 10:34:05.822 internal_send: 2405:8a00:4001:17:4c31::10f#43969: Invalid argument
18-Sep-2019 10:34:05.822 client @0x7ffa180e6f30 2405:8a00:4001:17:4c31::10f#43969 (d634f0139c3111a3f6189121da0f9c189a2b05a2268bfdde5a382a34.iith.ipv6.ernet.in): error sending response: invalid file

This problem occurs because of the ARP table overflow. To resolve this problem, we need to increase the table size to handle a large number of queries from IoT devices [23]:

echo 2048 > /proc/sys/net/ipv6/neigh/default/gc_thresh1
echo 8192 > /proc/sys/net/ipv6/neigh/default/gc_thresh2

b. By default, BIND name server is able to handle 100 TCP clients.

<TIME_STAMP> client: warning: client <IP>#<PORT>: no more TCP clients: quota reached

To increase the support for TCP clients, we have to specify the number of clients

in “named.conf” file [24].

c. When the DNS response size is more than 512 bytes, the DNS will not respond with UDP protocol. It will send a response with a flag bit on for “**DNS message is truncated**.” After sending this, DNS will fall back to TCP by default. If the UDP protocol is desired, we need to enable eDNS while sending the query. In this case, the response comes using UDP, and the fragmentation will happen at the source.

c. Simulation of Packet Loss

We simulated DNS packet loss on the path between emulated IoT client and the name server using ip6tables [25]. The ip6tables is a Linux kernel utility which is used to set up, maintain, and inspect tables of IPv6 packet filter rules. Each table contains several built-in chains and may also include user-defined chains. Each chain is a list of rules which can match a set of packets. The rules in each chain specify what to do with a packet that matches. We created a user-defined chain in emulated IoT devices as well as authoritative name server to simulate the packet loss. The command used is as follows,

sudo ip6tables -N TCP_UDP

At the emulated IoT device, the rule in the chain "TCP_UDP" should match DNS response, and at the authoritative name server, the rule should match DNS queries. As we know DNS response has the source port number 53, the following commands are used to create rules for filtering out the DNS responses at the emulated IoT client,

sudo ip6tables -A TCP_UDP -p tcp --sport 53 -j DROP
sudo ip6tables -A TCP_UDP -p udp --sport 53 -j DROP

The DNS requests have destination port number as 53, for creating the rule in order to filter out DNS requests the following commands are used,

sudo ip6tables -A TCP_UDP -p tcp --dport 53 -j DROP
sudo ip6tables -A TCP_UDP -p udp --dport 53 -j DROP

Finally, for simulating a packet loss for x% at the client-side and y% at the authoritative name server, the following command is used,

Client: *sudo ip6tables -A FORWARD -m statistic --mode random --probability <x> -j TCP_UDP*
Server: *sudo ip6tables -A INPUT -m statistic --mode random --probability <y> -j TCP_UDP*

The ip6tables drops the packets randomly with probability x% that matches the rules set up in TCP-UDP. If we simulate a packet loss of x% at both client-side and y% at server-side, the loss rate on the path can be calculated using the following equation,

$$Path\ loss\ rate = x + (1-x)y$$

In case of multiple retries by IoT client to get OX response due to losses on the path, theoretically the effective loss rate experienced by IoT client while sending OX queries and getting OX responses is calculated using the following equation,

$$effective\ loss\ rate = x + (1-x)y + (1-(1-x)y)x + \ldots.\ Till\ no.\ of\ retries$$

d. Simplified Update Logics

Each emulated IoT device runs a simplified variant of Update Logic #2 whose pseudocode is as follows,

1. **Packet Loss**
    a. *For i in range(0,500) /*Sending 500 OX queries to measure the average successful updation rate of IoT device*/*
        i. *Select a random version of the firmware in {1,2,3,4,5}*
        ii. *Send OX query to the name server*
        iii. *If OX query is successful*
            1. *Increment success_count*
            2. *Goto **step a***
        iv. *else*
            1. *If we tried for 6 times and still no success in getting OX reply,*
                a. *Increment failure_count*
                b. *Give up and go to **step a***
            2. *else*
                a. *Wait for a time interval selected by random binary exponential backoff*
                b. *Goto **step ii** after the expiry of time interval*

2. **Corrupted Firmware**
    a. *For i in range(0,500) /*Sending 500 OX queries to measure the average successful updation rate of IoT device*/*
        i. *Select a random version of the firmware in {h1,h2,h3,h4,h5}*
            / h indicates the hash value */*
            1. *queryString = <random_version>.domain_name*

ii. *Select a uniform random number r in [0,1]*
iii. *If r <= X /* X is the corrupted firmware rate */*
    1. *Select a dummy firmware version to simulate corrupted firmware*
    2. *queryString = <dummy_version>.domain_name*
iv. *Send OX query to the name server using queryString*
v. *If OX query is successful /* OX Reply is received */*
    1. *Increment success_count*
    2. *Goto **step a***
vi. *else*
    1. *If we tried for 6 times and still no success in getting OX reply,*
        a. *Increment failure_count*
        b. *Give up and go to **step a***
    2. *else if we get NXDomain error response*
        a. *queryString = <modelID>.domain_name*
        b. *Goto **step ii***
    3. *else*
        a. *Wait for a time interval selected by random binary exponential backoff*
        b. *Goto **step ii** after the expiry of time interval*

e. Performance Evaluation of IoT firmware Update Logic

At the authoritative name server, the zone file contains OX RRs. A sample of the OX RRs is shown in Table 6.

Table 6: A Sample Zone File with OX RRs

Domain Name	RR	EID	Type	Location	Media_type	Data
h1	OX	12	101	1	text/plain	URL of v2
	OX	12	102	1	text/plain	Hash value of v2
	<Other non mandatory fields like email address (type = 1), website (type = 2), contact number (type = 3), dependency with other firmwares (type = 105-200), description (type = 104)>.					
9811 [Default entry]	<Mandatory fields as above>					
9811 [Default entry]	<Non Mandatory fields as above>					

In all the experiments, details of five firmware versions are stored in the zone file with mandatory OX RRs. The latest firmware version is 5 and the oldest being version no is 1. From version 1, we can go to version 2. Version 2 is dependent on version 3. From versions 3 and 4, we can install version 5.

Scenario 1: Effect of Packet Loss

The goal of this experiment is to study the behaviour of the firmware update process (i.e., simplified Update Logic #2) when the underlying network is susceptible to packet losses. DNS has retry mechanism by default. Here, to study about the effect of tries count, default retry mechanism is disabled. Here, the assumption is that all the devices are not doing any sequential update. They are randomly selecting one of the version numbers and sending the OX query. If they are able to update to the next version specified in the zone file, it is considered as successful.

Each emulated IoT device sends 500 OX queries to the name server one after another. If the device gets the reply, it picks up the next query and continues the process. If the device did not get an answer in specified timeout value, it waits for a random amount of time picked up by binary random exponential backoff before retrying. If the emulated IoT device did not receive the reply even after six such tries, then we consider such an update as a failure.

The experiment is conducted for emulated IoT device counts of 1, 300 and 3000 by varying loss rate on the path between IoT clients and the authoritative name server from 10% to 75%. The result is shown in the plot Fig. 13. The plot also shows the ideal curve which is calculated based on the effective packet loss formula presented earlier. Here, the assumption is that the loss rates at both client and server side are equal; i.e. x=y. Experimental results in Fig. 13 show that results with 1 device, 300 devices, and 3000 devices are overlapping with the ideal curve. As we can see, even if the packet loss is 75%, we are able to achieve 82% successful updates using retry mechanism in the proposed update logic. For each device count, we have also calculated the average number of tries as shown in Fig. 14. As it can be seen in graph that the average number of tries are independent of the number of devices. Increase in packet loss causes frequent packet dropping which results in an increase in the number of retries. Here, in all the experiments, the maximum number of tries has been set as 6. So, in the maximum of 6 tries, if the client is not getting response, that attempt is considered as failure case.

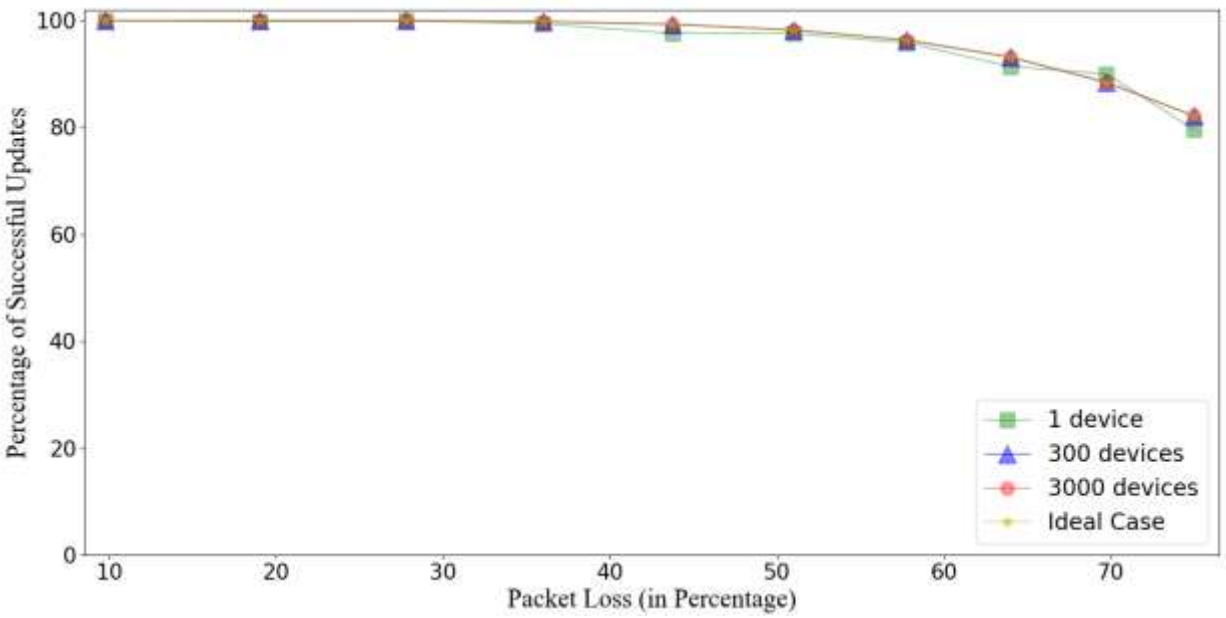

Fig. 13: Percentage of Successful Update vs Packet Loss

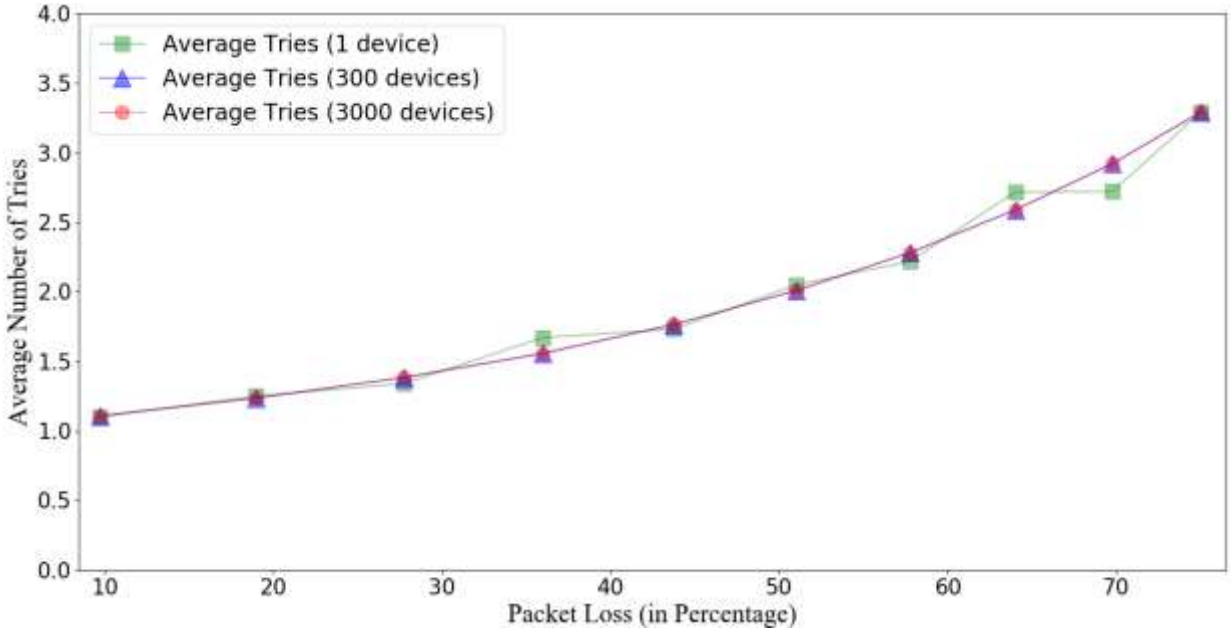

Fig. 14: Average Number of Tries vs Packet Loss

Scenario 2: Single vs Multiple Fragments

If the response size is more than 1500 Bytes, the fragmentation will happen at the source. To study the behavior in the case of multiple fragments, non-mandatory OX records are used to increase the size of the OX response in this experiment. In the case of fragmentation, TCP protocol is used by

default. By enabling eDNS, it sends all the fragments into one packet using the underlying protocol as UDP.

Here, we can see in Fig. 15, the number of successful counts in case of 2 fragments case is overlapping with that of 1 fragment case. Even though there are overlapping, as we can see in Fig. 16, the number of tries count in case of 2 fragments case is slightly higher than that in case of 1 fragment case. There will be more packet losses because if one fragment is lost, the response will be considered as lost and then the client has to retry.

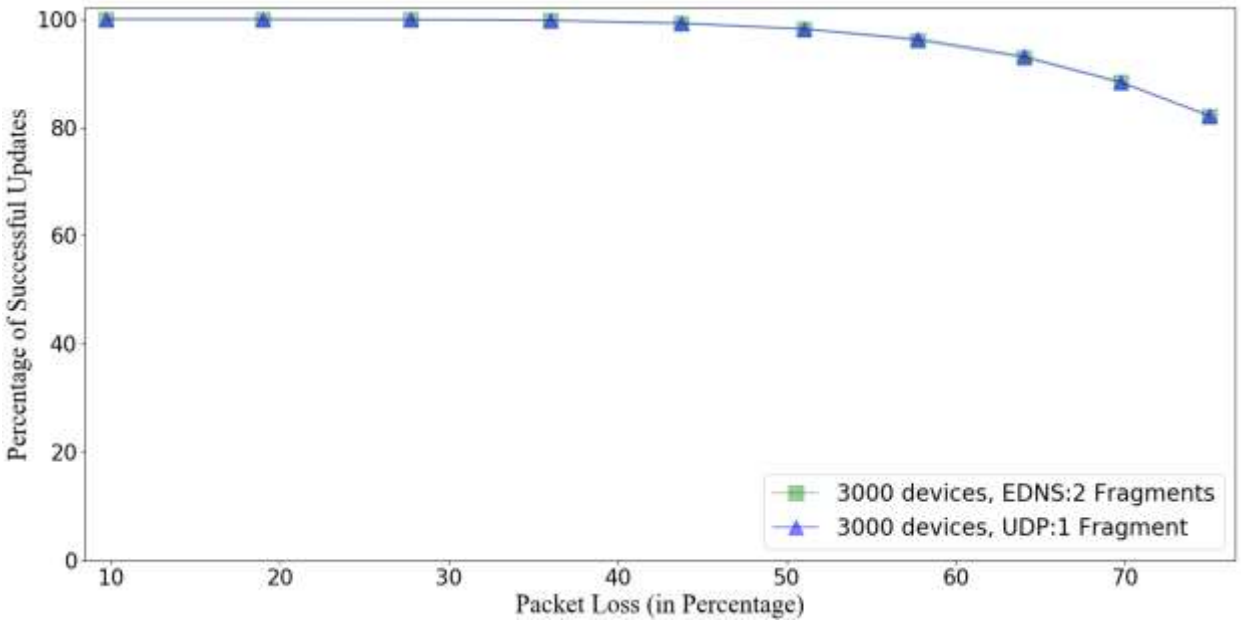

Fig. 15: Percentage of Successful Updates vs Packet Loss

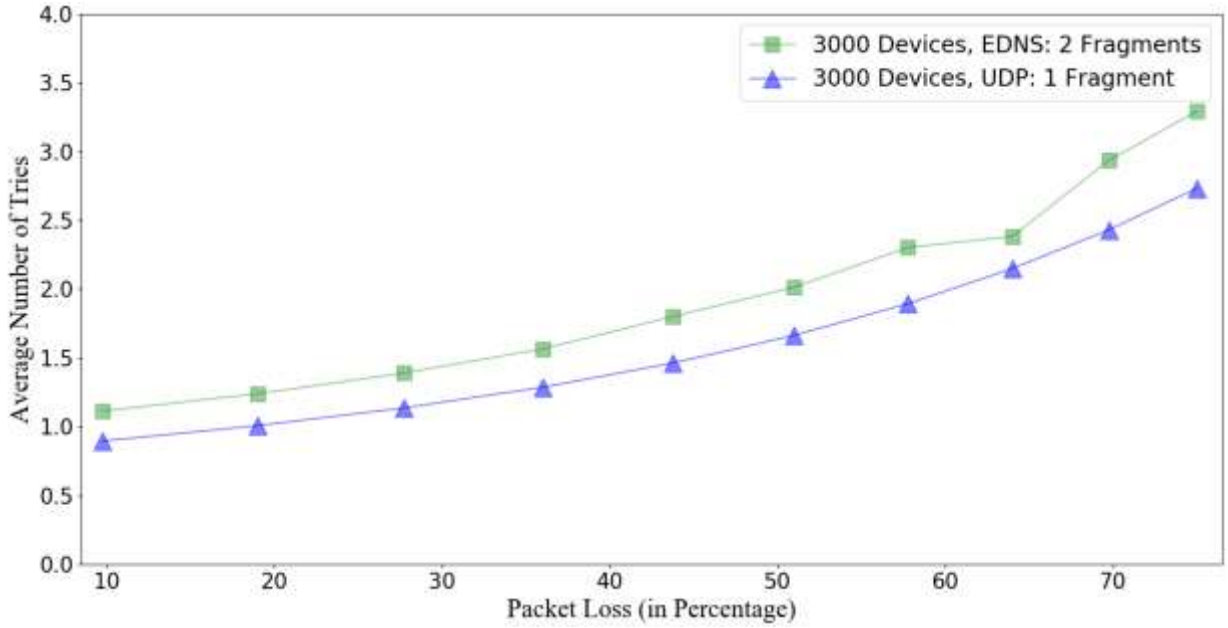

Fig. 16: Average Number of Tries vs Packet Loss

Scenario 3: Effect of Corrupted Firmware

In this scenario, the emulated IoT devices send 500 queries one after another to the name server seeking for OX resource records by randomly selecting a firmware version like in Scenario 1. To study the behaviour of the update process when the client is running corrupted firmware, we emulate the corrupted firmware as specified in the simplified update logic.

Each IoT device at the beginning of each query generates a uniform random number in the range [0, 1]. To emulate 10% corrupted firmware scenario for example, we check whether the random number generated is less than or equal to 0.1. If the generated random number is less than 0.1, then instead of selecting a valid hash value, the IoT client will query with some wrong hash value, which is an indication that firmare at the client is corrupted. Here, our assumption is that even though the firmware at client is corrupted, the client can still query for OX RRs from the name server. Name server on the other hand receives a query for which it doesn't have any matching OX RRs, so it sends NXDomain error response to the client. When the client gets NXDomain error response, it queries with Model ID to get OX response. When the device queries with Model ID, it gets default OX RRs as the OX response which contains URI of the firmware that can directly be installed on the given Model ID without any dependency issues. Even before sending the query with Model ID, there can still be corruption cases possible.

The simplified update logic as mentioned in subsection d is used for running this experiment. We calculated the percentage of successful firmware update at various corrupted firmware rates starting from 10% to 100%. Like other scenarios, the try value is 6 here, which means IoT clients will try for maximum of 6 times. If still they receive NXDomain error, then such update is considered as failure. The plot by varying corrupted firmware rates on x-axis and measuring the successful percentage on Y-axis is shown in Fig. 17. From the plot, it is clear that, even though we have a corrupted firmware rate as 90%, the percentage of successful update is 45%. We have also plotted average number of tries for various corrupted firmware rates as shown in Fig. 18.

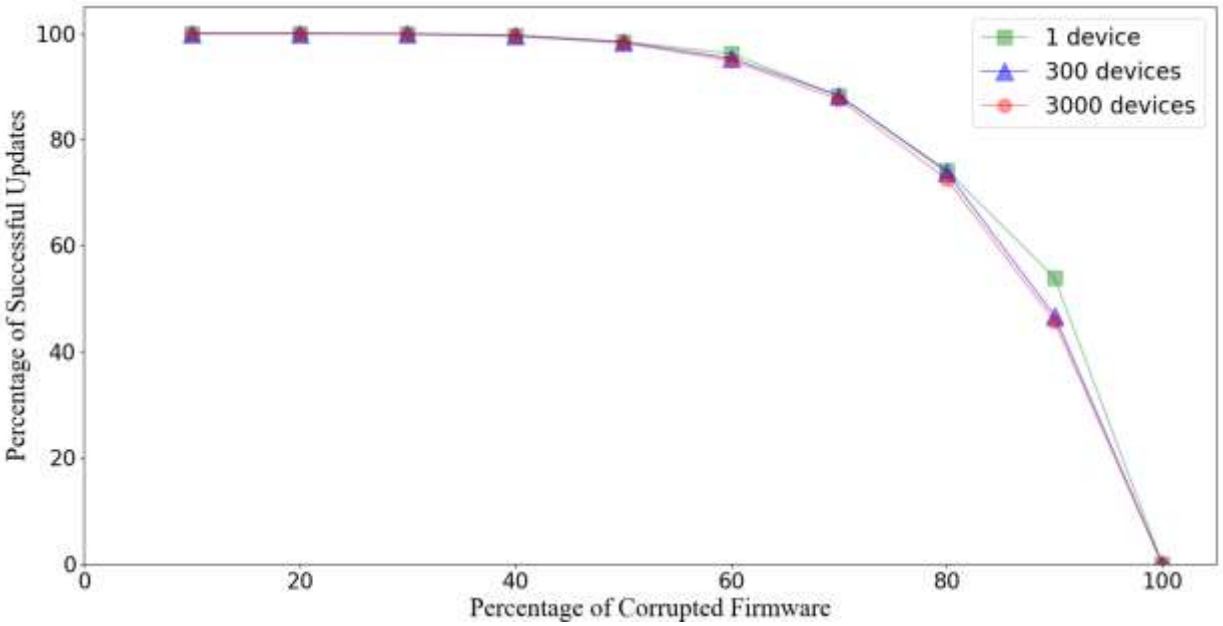

Fig. 17: Percentage of Corrupted Firmware vs Percentage of Successful Updates

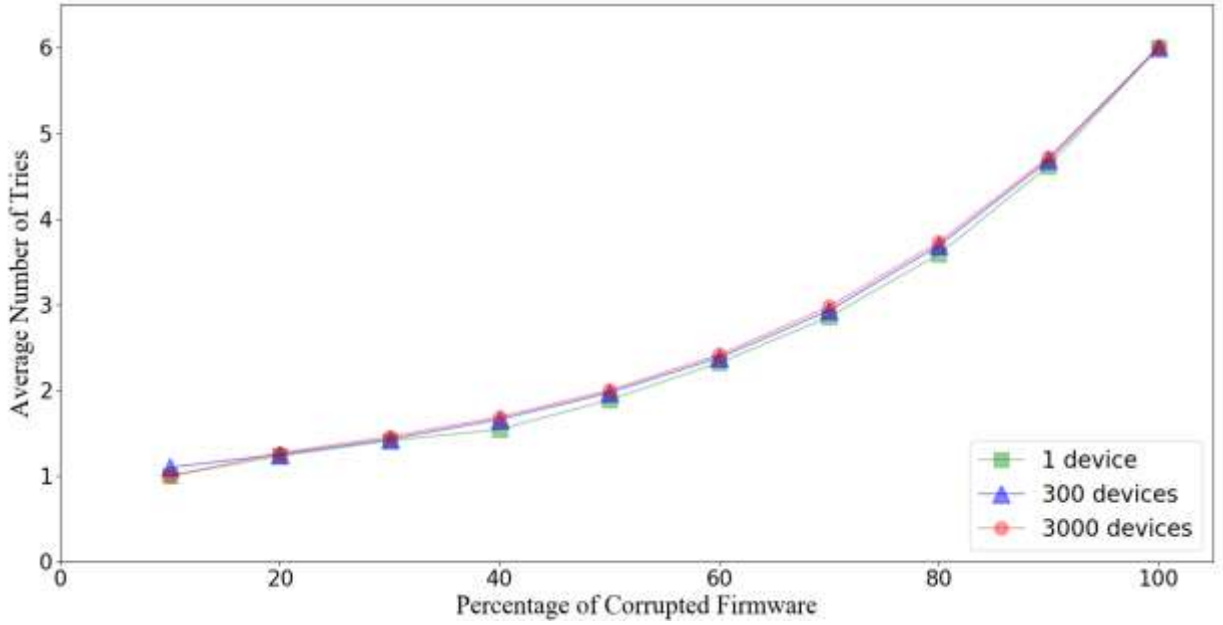

Fig. 18: Percentage of Corrupted Firmware Average Number of Tries

Scenario 4: Effect of nsupdate

In this experiment, nsupdate client sends nsupdate requests to the name server using dnsperf at 2500 updates per second infinitely. Dnsperf [26] is used here to send nsupdate requests to the name server. When the update is going on, dnsperf client queries for the same record which is being updated. Response per second with nsupdate and without nsupdate by varying updates per second is as shown in Fig. 19. The response per second received when nsupdate is going on is less than

the response per second without nsupdate.

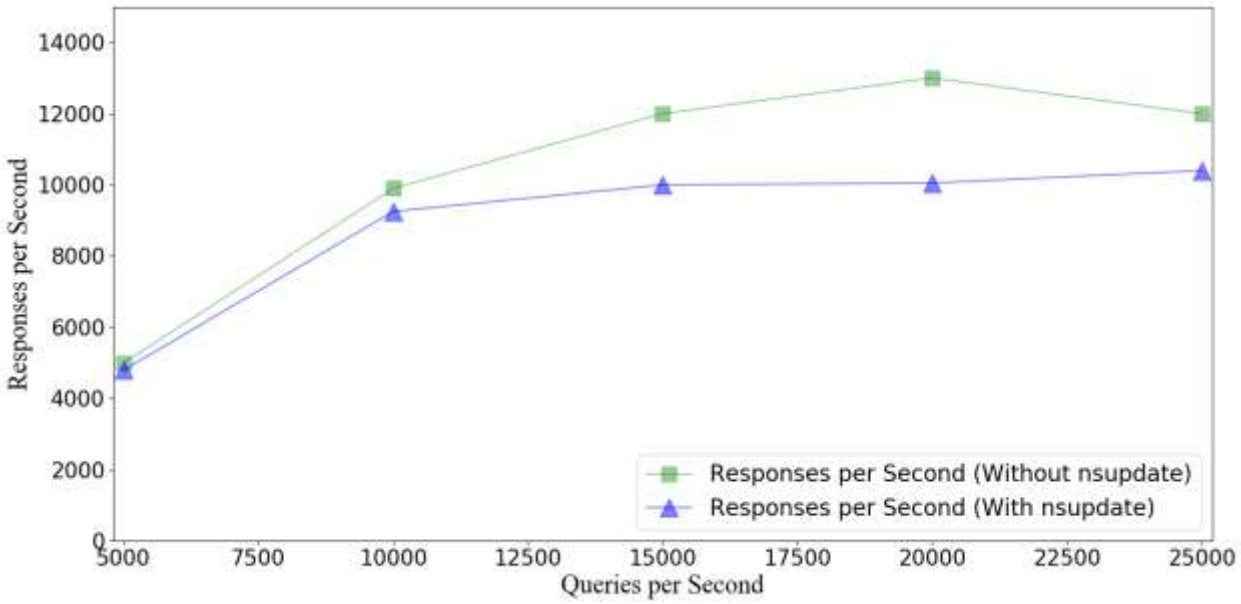

Fig. 19: Queries per Second vs Responses per Second

IX. Conclusions

This report examines the appropriateness and scalability of using DNS infrastructure to reliably update IoT device firmware by conducting experiments in various adverse network circumstances. For an IoT device to successfully upgrade to the latest version, we have come up with a lightweight update logic which uses the hash value of the firmware version to guard against Man-In-The-Middle (MITM) attacks. In this report, four different variants of update logics are proposed which are suitable for different kinds of applications. The update logic #1 is the most simplified update logic, where the client sends the current version it is running and gets the OX RRs containing the URI of the next version from the name server. The update logic #2 provides an additional lightweight security mechanism on top of update logic #1, by using hash value of the firmwares instead of simply using the version number while sending queries to the name server. Using hash value serves two purposes, first one is if we are using hash value of the firmware version instead of actual firmware version, we are not revealing the firmware versions the client is running. Another advantage is if we are using hash value, the corruption between the IoT device and OEM server can also be detected. The next update logic is built on top of update logic #2, which defines a new type field called criticality of firmware which helps the IoT devices to get the firmwares using lesser number of updates. This update logic is more suitable for those devices which have resource constraints like limited bandwidth and computational resources. The update logic #4 is defined to handle multipart firmwares which are dependent on each other.

We have evaluated our testbed containing name server, emulated IoT devices, nsupdate client and firmware server in an IPv6 only network. We have considered various conditions such as packet loss, corrupted firmware, handling simultaneous update and queries and multiple fragments by varying the number of emulated IoT devices between 1 and 3000. Our experimental results show that the name server serves the request from 3000 devices like the way it handles requests from one device i.e., name server can handle the additional load by 3000 devices querying for OX RRs. Our observations of packet loss reveal that even though we have packet loss of 75% in the network, the update logic works with a success rate of 82%. We observe 6 retries for single fragment case. In the case of two fragments, average number of retries is a little higher. A study on client running corrupted firmware shows that, when 90% of the time client runs corrupted firmware, we are still able to get a success rate of 45%. We further conclude that the manufacturer can also update the OX RRs present in the name server with a very high update rate while the clients are simultaneously querying for the OX RRs.

References:

[1] Juniper Research. [n.d.]. *IOT CONNECTIONS TO GROW 140% TO HIT 50 BILLION BY 2022, AS EDGE COMPUTING ACCELERATES ROI* https://www.juniperresearch.com/press/press-releases/iot-connections-to-grow-140pc-to-50-billion-2022

[2] Cisco. 2017. *Cisco Visual Networking Index: Forecast and Trends, 2017 - 2022* White Paper.

[3] The Guardian, 2017 Aug, *Hacking risk leads to recall of 500,000 pacemakers due to patient death fears*, https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update

[4] Allot. 2019. *Silex Malware: Deadly New Virus Bricks 1000s of IoT Devices.* https://www.allot.com/blog/silex-malwar

[5] ICANN meeting, 2017 https://icann60abudhabi2017.sched.com/event/CbHh/joint-meeting-icann-board-technical-experts-group-teg

[6] P. Mockapetris. 1987. Domain Names - Implementation and Specification, https://www.ietf.org/rfc/rfc1035.txt

[7] P. Nikander. 2008. *Host Identity Protocol (HIP) Domain Name System (DNS) Extension.* RFC 5205. https://tools.ietf.org/html/rfc5205

[8] B. Niven-Jenkins. 2012. *Content Distribution Network Interconnection (CDNI) Problem Statement.* RFC 6707. https://tools.ietf.org/html/rfc6707

[9] P. Hallam-Baker. 2013. *DNS Certification Authority Authorization (CAA) Resource Record DNS Certification Authority Authorization (CAA) Resource Record.* RFC 6844. https://tools.ietf.org/html/rfc6844

[10] J. Jeong. 2010. *IPv6 Router Advertisement Options for DNS Configuration.* RFC 6106. https://tools.ietf.org/html/rfc6106

[11] E. Rye. 2019. *Customer Management DNS Resource Records.* RFC 8567.

https://tools.ietf.org/html/rfc8567
[12] J. Damas. 2013. *Extension Mechanisms for DNS (EDNS(0))* https://tools.ietf.org/html/rfc6891
[13] A. Durand. 2017. *DOA over DNS.* https://tools.ietf.org/html/draft-durand-doa-over-dns-02
[14] S. Sun. 2003. *Handle System Overview.* https://tools.ietf.org/html/rfc3650
[15] N. Freed. 2013. *Media Type Specifications and Registration Procedures* https://tools.ietf.org/html/rfc6838
[16] B. Moran. 2017. *A Firmware Update Architecture for Internet of Things Device,* https://tools.ietf.org/html/draft-moran-suit-architecture-00
[17] P. Vixie. 1997. *Dynamic Updates in the Domain Name System (DNS UPDATE).* RFC 2136. https://www.ietf.org/rfc/rfc2136.txt
[18] P. Vixie. 2000. *Secret Key Transaction Authentication for DNS (TSIG),* RFC2845. https://www.ietf.org/rfc/rfc2845.txt
[19] Django Framework, https://github.com/django/django
[20] Ansible, https://github.com/ansible/ansible
[21] BIND9, https://github.com/isc-projects/bind9
[22] DNSPython, https://github.com/rthalley/dnspython
[23] ARP table overflow, https://packetfence.org/support/faq/solving-neighbour-table-overflow-errors-large-subnets.html
[24] TCP client support, https://access.redhat.com/solutions/264683
[25] Iptables, https://github.com/wertarbyte/iptables
[26] DNSPerf, https://github.com/DNS-OARC/dnsperf
[27] Firmware update demonstration using NodeMCU, https://github.com/iot-linti/doa-sketchs/
[28] Patched ESP8266 library to support OX resource record, https://github.com/iot-linti/Arduino-esp8266/tree/doa