DESIGN AND DEVELOPMENT OF AN INTERNET-OF-THINGS (IoT) GATEWAY FOR SMART BUILDING APPLICATIONS

ADITYA NUGUR

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

In

Electrical Engineering

Saifur Rahman, Chair

Manisa Pipattanasomporn

Jaime De La Ree lopez

29th September, 2017
Arlington, VA

Keywords: IoT, Smart Buildings, IoT Gateway, Building Energy Management

DESIGN AND DEVELOPMENT OF AN INTERNET-OF-THINGS (IoT) GATEWAY FOR SMART BUILDING APPLICATIONS

ADITYA NUGUR

ABSTRACT

With growing concerns about global energy demand and climate change, it is important to focus on efficient utilization of electricity in commercial buildings, which contribute significantly to the overall electricity consumption. Accordingly, there has been a number of Building Energy Management (BEM) software/hardware solutions to monitor energy consumption and other measurements of individual building loads. BEM software serves as a platform to implement smart control strategies and stores historical data. Although BEM software provides such lucrative benefits to building operators, in terms of energy savings and personalized control, these benefits are not harnessed by most small to mid-sized buildings due to the high cost of deployment and maintenance. A cloud-based BEM system can offer a low-cost solution to promote ease of use and support a maintenance-free installation.

In a typical building, a conventional router has a public address and assigns private addresses to all devices connected to it. This led to a network topology, where the router is the only device in the Internet space with all other devices forming an isolated local area network behind the router. Due to this scenario, a cloud-based BEM software needs to pass through the router to access devices in a local area network. To address this issue, some devices, during operation, make an outbound connection to traverse through the router and provide an interface to itself on the Internet. Hence, based on their capability to traverse through the router, devices in a local area network can be distinguished as cloud and non-cloud devices. Cloud-based BEM software with sufficient authorization can access cloud devices. In order to access devices adhering to non-cloud protocols, cloud-based BEM software requires a device in the local area network which can perform traversal through the router on behalf of all non-cloud devices. Such a device acts as an IoT gateway, to securely interconnect devices in a local area network with cloud-based BEM software.

This thesis focuses towards architecting, designing and prototyping an Internet-of-Things (IoT) gateway which can perform traversal on behalf of non-cloud devices. This IoT gateway enables cloud-based BEM software to have comprehensive access to supported non-cloud devices. The IoT gateway has been designed to support BACnet, Modbus and HTTP RESTful, which are the three widely adopted communication protocols in the building automation and control domain. The developed software executes these three communication protocols concurrently to address requests from cloud-based BEM system. The performance of the designed architecture is independent of the number of devices supported by the IoT gateway software.

DESIGN AND DEVELOPMENT OF AN INTERNET-OF-THINGS (IoT) GATEWAY FOR SMART BUILDING APPLICATIONS

ADITYA NUGUR

GENERAL AUDIENCE ABSTRACT

Building energy management (BEM) software is developed to manage smart devices deployed in commercial buildings. Conventional building energy management systems are hosted on hardware systems and operate within building vicinity. Being physically installed, conventional BEM software performance is limited by deployed hardware specifications and is prone to building intrusions.

Cloud technology is recently developed paradigm which promotes hardware independent software deployment. A cloud-based building energy management software would, therefore, outperform any conventional BEM software installation. Although beneficial by being remotely deployed, cloud-based BEM software lacks direct device connectivity. Hence for accessing devices, a cloud-based BEM software requires the devices to support remote connectivity. Support for remote connectivity by a device depends on the communication mechanism adopted by the device. In a typical building, the majority of devices don't support remote connectivity.

As a solution to this problem, this thesis focuses on developing an Internet of Things (IoT) gateway software, which is hosted on the building vicinity to act as a proxy for accessing devices. An open architecture IoT gateway prototype is designed which is scalable to support any protocol. Developed prototype platform supports eleven devices of both industrial and next-generation communication protocols. Although deployed on hardware resource, the software is designed to use the minimum amount of RAM for its operation. Developed IoT gateway software can, hence, resolve the feasibility issue of cloud-based BEM software.

Acknowledgements

I am very thankful to Dr. Saifur Rahman for providing me an opportunity to pursue my graduate studies at Virginia Tech. Also, I express my sincere gratitude to Dr. Murat and Dr. Manisa for their continuous support and assistance provided by them during my stay at Advanced Research Institute (ARI). I thank Dr. De La Ree and Dr. Broadwater for their support during my stay at Blacksburg.

Doing a master's program requires delivery of significant contribution in a short-term commitment. This requires demonstration of strong background Knowledge and skills. Rajendra, Xiangyu, Mengmeng, Avijit were really supportive at ARI and gave me a boosted start without which this thesis would never be possible.

I thank my parents and all my friends who supported even during my bad times and guided me towards my career goal.

Table of Contents

1. INTRODUCTION 1
1.1 BACKGROUND 1
1.2 NEED FOR IOT GATEWAY 2
1.3 THESIS STATEMENT AND SCOPE 2
1.4 CONTRIBUTIONS 3
1.4.1 Secure and Reliable connectivity with LAN devices 3
1.4.2 Unified platform for BACnet, Modbus and HTTP RESTful protocols 3
1.4.3 Comma Separated Values File (csv) based API development for BACnet and Modbus protocols 3
2. BACKGROUND AND LITERATURE REVIEW 4
2.1 BACKGROUND 4
2.1.1 Concept of Smart Grid and Smart City 4
2.1.2 Building Energy Management Software 5
2.1.3 Cloud Technology 9
2.1.4 Interconnecting BEM software, IoT and Cloud technology 13
2.1.5 Challenges for Cloud Architecture 14
2.1.6 Concept of IP Addressing and Multi-Networking 14
2.1.7 Challenges Due to Non-Standardized NAT Architecture 16
2.1.8 NAT Traversal Techniques 17
2.1.9 Choosing APT NAT Traversal 20
2.1.10 Non-Cloud and Cloud Protocols 20
2.1.11 Concept and Role of the IoT Gateway 30
2.2 LITERATURE REVIEW 30
2.2.1 Gateway as device management software 30
2.2.2 Gateway as proxy device 31
2.2.3 Commercial IoT gateway products 31
2.3 RESEARCH GAP 31
3. IoT GATEWAY SOFTWARE FEATURES, ARCHITECTURE AND DESIGN 33
3.1 IoT GATEWAY FEATURES 33
3.1.1 NAT traversal and secure communication with the cloud-based BEM software 33
3.1.2 Protocol translators to handle requests 33
3.1.3 Software Development Approach 35

3.2 ARCHITECTURE AND DESIGN 35
3.2.1 Software Architecture 35
3.2.2 Concurrency Model 39
3.2.3 Software Design 42
4. IoT GATEWAY PROTOTYPE DEVELOPMENT AND PERFORMANCE TESTS 54
4.1 IoT GATEWAY PROTOTYPE DEVELOPMENT 54
4.2 ADDING NEW PROTOCOLS AND DEVICES 56
4.3 PERFORMANCE TESTS 56
4.2.1 Idly running 58
4.2.2 IoT gateway software running one protocol actor and one device 58
4.2.3 IoT gateway software running one protocol actor and 10 devices 58
4.2.4 IoT gateway software running one protocol actor and 40 devices 59
4.2.5 IoT gateway software running two protocol actors and ten devices each 59
4.2.6 IoT gateway software running two protocol actors and 40 devices each 60
4.2.7 IoT gateway software running three protocol actors and ten devices each 60
4.2.8 IoT gateway software running with three protocol actors and 40 devices each 60
4.3 PERFORMANCE COMPARISON 61
5. CONCLUSION AND FUTURE WORK 62
REFERENCES 64

List of Figures

Figure 1: Communication between two services in terms of OSI model 7
Figure 2: A typical IP packet 14
Figure 3: A typical local area network topology 15
Figure 4: Frame format of a Modbus RTU with ADU and PDU constituents 22
Figure 5: Typical Modbus RS-485 topology 23
Figure 6: MODBUS TCP/IP with MBAP header frame 23
Figure 7: BACnet architecture in terms of OSI model 25
Figure 8: Topology of IP tunneling among BACnet device 27
Figure 9: Topology of message transfer among BACnet/IP devices.. 27
Figure 10: Cloud-based BEM software architecture in conjunction with IoT gateway 32
Figure 11: Layout of Cloud-based BEM software along with IoT gateway................................ 35
Figure 12: Typical actor based model.. 41
Figure 13: Software Architecture of designed IoT gateway 42
Figure 14: Laboratory setup of the IoT gateway along with few non-cloud devices 55
Figure 15: RAM usage vs. number of processes running.. 61

List of Tables

Table 1: IOT platforms 12
Table 2: Private IP range.. 15
Table 3: Types of Network Address Translation performed by router.. 16
Table 4: Comparison of two major variants of Modbus protocol w.r.t standard OSI model 22
Table 5: Request- Response example 23
Table 6: Various LANs supported by BACnet protocol... 25
Table 7: Comparison of protocols which support cloud 29
Table 8: Comparison of various concurrency approaches... 38
Table 9: Comparison between TCP and UDP 44
Table 10: Comparison between pub-sub and req-resp pattern... 48
Table 11: Comparison between CoAP, RESTful HTTP and Web Socket 48
Table 12: Modbus read request function codes 52
Table 13: Modbus write request function codes 52
Table 14: Devices currently supported by IoT-Gateway 55
Table 15: RAM consumption by software when idle 58
Table 16: RAM consumption by software when one protocol actor is monitoring one device ... 58
Table 17: RAM consumption by software when 1 protocol actor is monitoring 10 devices 59
Table 18: RAM consumption by software when 40 devices are monitored by 1 protocol actor . 59
Table 19: RAM consumption by software with two protocol actors and 10 devices each........... 59
Table 20: RAM consumption by software when 80 devices are monitored................................. 60
Table 21: RAM consumption by software when 30 devices are monitored with 3 protocol actors
.... 60
Table 22: RAM consumption by software when 120 devices are monitored............................... 60
Table 23: Comparing RAM usage in various scenarios 61

Nomenclature

ADU	-	Application Data Unit
AES	-	Advanced Encryption Standard
ALG	-	Application Layer Gateway
API	-	Application Program Interface
ARCNET	-	Attached Resource Computer NETwork
BACnet	-	Building Automation and Control Networks
BBMD	-	BACnet Broadcast Management
BEM	-	Building Energy Management
CoAP	-	Constrained Application Protocol
CPU	-	Computer Processor Unit
DCS	-	Distributed Control System
DES	-	Data Encryption Standard
DNS	-	Domain Name Server
GB	-	GigaByte
GIL	-	Global Interpreter Lock
GPRS	-	General Packet Radio Service
HTML	-	HyperText Markup Language
HTTP	-	HyperText Transfer Protocol
ICE	-	Interactive Connectivity Establishment
IGDP	-	Internet Gateway Device
IOT	-	Internet of Things
IP	-	Internet Protocol
IPC	-	Inter-Process Communication
JSON	-	Javascript Object Notation
LAN	-	Local Area Network
MAC	-	Message Authentication Code
MB	-	MegaByte
MBAP	-	ModBus Application
MIME	-	Multipurpose Internet Mail Extensions
MIT	-	Massachusetts Institute of Technology
MW	-	Mega Watt
NAPT	-	Network Address and Port Translation
NAT	-	Network Address Translation
OPC	-	OLE for Process Control
OS	-	Operating System
OSI	-	Open Systems Interconnection
PAAS	-	Platform As A Service
PAT	-	Port Address Translation
PCB	-	Printed Circuit Board
PCP	-	Port Control Protocol

PDU	-	Protocol Data Unit
PLC	-	Programmable Logic Control
PMP	-	Port Mapping Protocol
PTP	-	Point to Point
PV	-	Photo Voltaic
RAM	-	Random Access Management
REST	-	Representational state transfer
RFC	-	Request for Comments
RTU	-	Roof Top Unit
SAAS	-	Software As A Service
SCADA	-	Supervisory control and data acquisition
SOCKS	-	Socket Secure
SSL	-	Secure Socket Layer
STUN	-	Session Traversal Utilities for NAT
TCP	-	Transmission Control Protocol
TLS	-	Transport Layer Security
TPM	-	Trusted Platform Module
TURN	-	Traversal Using Relays around NAT
UDP	-	User Datagram Protocol
UPnP	-	Universal Plug and Play
URI	-	Uniform Resource Locator
VAV	-	Variable Air Volume
WAN	-	Wide Area Network
XML	-	eXtensible Markup Language
XOR	-	Exclusive OR

1. INTRODUCTION

1.1 BACKGROUND

An electric power network is an extremely complex system, which is immensely large and needs to be uninterruptedly functional. Large-scale electrical outages will result in loss of revenue, a high level of discomfort for the consumers and hamper the progress of all other interlinked industrial sectors. A moderate-sized power system with a peak load of 10,000 MW and having an annual load factor of 60% is expected to have an annual fuel cost of 4,993 Million dollars [1]. Thus even a small percentage of energy savings results in millions of dollars saving annually.

To improve the performance and efficiency of the current power grid, it is essential to analyze the scenario and ascertain gist of the electric grid. The entire electric grid layout can be divided into three domains namely generation, transmission and distribution. All power generated is an attempt to meet the demand at the distribution level, thereby leading to a direct causal relationship between energy consumed and power produced. This causal dependence of power generation, transmission domain with the distribution domain signifies the vitality of distribution domain in the whole picture. Ascribing to the above fact, the best approach to decrease power generation is by making efficient power consumption decisions at a micro level to accumulate a macro level energy savings. Implementing such energy efficient schemes can prove to be most fruitful to make a large-scale impact on the electric grid utility.

Residential and commercial customers contribute to 40 % of the total power consumption. Hence, efficient load management at the level of building scale with residential consumers as an endpoint would have a more significant contribution to overall energy savings. To initiate efficient energy consumption at the consumer endpoint, endpoint users need to be trained or realized the worth of the energy being consumed. Traditionally, this message of efficient energy consumption is disseminated to end users by costing end users to pay as per the amount of power they consumed. In many cases, most of the corporate buildings or public buildings such as libraries, schools, etc., have end users who are not penalized for power usage. This breaks the sole purpose of inculcating efficient energy consumption attitude on end users. In such scenarios, it would be the responsibility of building administrators to have control over the amount of power being consumed. This led to the development of smarter building loads which revolutionized traditionally developed passive power consuming devices. The conventional power consuming devices are made smarter by incorporating multi-disciplinary technologies through which they can store energy usage information and communicate its knowledge to other devices. This smart revolution is an implementation of the broader term “Smart Grid.”

Aside from machine-to-machine communications, software systems with intelligent algorithms were developed to smoothly interface the machine domain with human perception. Such software’s provide a user interface for an operator to monitor and implement smart control strategies on smart devices. There are many such building energy management (BEM) software resources available commercially in the market. These are developed by Johnson Controls, Siemens, Schneider Electric, etc. Their installation cost 100K’s of dollars, thus, deeming them to be commercial resources, which are affordable to large-sized buildings with substantial investments. Additionally, smart control on certain building loads required the installation of proprietary devices and attention to interoperability with existing infrastructure. These requirements for additional efforts to adopt BEM environment led to a negative perception and resulted in lower rates of BEM software adoption among non-profit medium-sized buildings, such as libraries, schools, offices, etc.

The early 2000s witnessed a trend in growth of community collaborations to develop open source software leading to a steep growth and evolution in the field of software development. With the advent of open source communities and consequent growth of technology, many open source BEM software platforms were developed for non-profit usage. Additionally, advancements in communication technologies promoted the easy development of smart devices, even for a small to the moderately ranged company. These intelligent devices are developed along with documented Application Program Interfaces (APIs) to access the device from any compatible open platforms, thereby addressing the proprietary protocol problem. Although mentioned breakthroughs fostered easy adoption of BEM solutions by addressing the cost of deployment, interoperability, open architecture, etc., BEM solution still has strong deterrents due to its interdisciplinary base. Due to its multidisciplinary correlation, BEM has interoperability, stability/reliability, ease of use, security, scalability, etc., issues.

To address interoperability, the developed BEM software must be modular to incorporate different protocols under the same hood. Additionally, to address scalability, BEM software requires adding support to any number of devices without performance degradation. Such a need for a scalable platform is a prevalent problem in the domain of software engineering. For being scalable, a software solution must have access to scalable physical hardware resources with high maintenance standards. This led to the advent of cloud computing in which computation and software maintenance is taken care at a remote end where scalable hardware resources are maintained. Replicating this, a new paradigm featuring a cloud-based BEM software can be developed which can be both easy to deploy and highly scalable due to unlimited resources. Such a cloud architecture has the capability to run as a single instance and maintain multiple buildings simultaneously. Additionally, with more number of input features from different buildings, cloud-based BEM software can implement even advanced control strategies. Such a framework would be the first step of progression towards a much broader domain "Smart Cities."

1.2 NEED FOR IOT GATEWAY

Although the cloud architecture would streamline BEM software adoption, it still suffers from a fundamental flaw, i.e., the feasibility of the cloud solution. Cloud-based BEM requires devices in the local area network to support remote access. Although recently developed smart devices are adaptive to cloud architecture by residing in Wide Area Network (WAN), there are still plenty of devices which are connected to local area network and can be controlled only by being on the wire. A cloud-based BEM software residing on a WAN lacks connectivity to such LAN based smart devices.

For a cloud-based BEM software to have comprehensive control on all devices in the building, it has to circumvent any network specific issues and should be able to access any device irrespective of the network the device is connected to. For this, cloud-based BEM software requires a proxy within a local area network, to rely on its communication to the devices which are local to the network. Such a connection with proxy should be secure and promote cloud-based architecture benefits. Hence, this proxy device should act like as an IoT gateway which is scalable to support many devices concurrently, modular to support multiple protocols and establishes a secure connection with cloud-based BEM software to enhance its functionality.

1.3 THESIS STATEMENT AND SCOPE

The objective of this thesis is to architect, design and demonstrate a prototype IoT gateway which can securely connect a cloud-based BEM software with local area network. For this, a brief introduction of conventional BEM software and recent technological advancements on which a traditional BEM software application can leverage is surveyed. These advancements include cloud technologies and the Internet of Things (IoT) era with a plethora of smart devices deployed. Later, operational principles of various communication protocols are analyzed in detail, to ascertain the feasibility of a cloud-based BEM in terms

of device connectivity. Based on different network topologies by which a traditional router isolates devices from the Internet, an IoT gateway is proposed which can perform on any network topology.

Following specific tasks have been performed in this thesis:

1) Developed an IoT gateway software architecture which maintains a secure persistent connection with a cloud-based BEM software. This software relays requests from the cloud-based BEM software to the devices in a local area network.

2) Developed an IoT gateway software platform that is scalable to support any protocol. For the prototype demonstration, BACnet, Modbus and RESTful—the three most commonly used building automation and control protocols are implemented.

3) Developed BACnet, Modbus API interfaces as a framework to which new devices can be integrated by adding device specific csv modules.

4) Performance of the developed IoT gateway platform was tested. Findings indicated that the developed IoT gateway consumed only 200 MB of RAM for monitoring 150 devices at one-minute monitoring time interval.

1.4 CONTRIBUTIONS

1.4.1 Secure and Reliable connectivity with LAN devices

This thesis addresses the fundamental issue which limits the wide adoption of a cloud-based BEM system. The developed IoT gateway software works in conjunction with cloud-based BEM software to discover, monitor and control devices in local area network. For this, IoT gateway software performs network address translation (NAT) traversal through a technique called hole punching. By performing NAT-traversal, it can identify itself to the cloud-based BEM software with a public IP address. Once connected to a cloud-based BEM, it maintains a persistent connection with cloud-based BEM software and relays user requests to the devices in local area network.

1.4.2 Unified platform for BACnet, Modbus, and HTTP RESTful protocols

BACnet and Modbus are two most widely adopted legacy protocols in the domain of building automation and controls. With the advent of cloud technologies, a number of next-generation smart devices based on RESTful application program interface (API) are deployed in the market. This thesis identifies the research gap of developing a platform which can smoothly integrate smart devices along with devices adhering legacy protocols.

1.4.3 Comma Separated Values File (csv) based API development for BACnet and Modbus protocols

Modbus and BACnet protocol frameworks are developed as API interfaces to which new devices can be integrated as csv based modules, thereby, promoting the easier integration of new BACnet and Modbus devices.

2. BACKGROUND AND LITERATURE REVIEW

2.1 BACKGROUND

2.1.1 Concept of Smart Grid and Smart City

Trending towards continuous advancement, the conventional electric system is in continuous modernization by leveraging on interdisciplinary technologies. This modernization paradigm focuses on transforming passive infrastructure into a responsive one. This approach to incorporate responsiveness and intelligence to the grid is termed as Smart Grid. The objective of developing smart grid is to envision an ecosystem where utility provider delivers service at minimum cost, in the smallest time possible, and simultaneously provide other critical services. Such a proactive power grid can increase reliability and decrease energy wastage. To modernize the grid to be proactive, it is essential to empower outdated power grid technology with "smart" technologies. This requires adoption to many cross-disciplinary technologies and strategies for smooth grid transition. In [2] author described smart grid to be based on three pillars. These are

1) Smart consumer
2) Smart utility
3) Smart market

Although there is a tight interdependency among all three pillars, it is intuitive that the pillar which is focused on the end user (Consumer) is most influential since remaining pillars are causally related to it. To augment smartness or pro-activeness at the consumer level, power consuming devices should have an inbuilt capability to record their behavior so that they can behave as a knowledge base rather than a passive load. Based on the recorded knowledge, consumer load can be modeled to implement smart control strategies. In order to achieve these objectives, smart grid should adapt to advancements in many interdisciplinary fields, such as data processing, transmission, analysis, communication technologies, etc.

With the beginning of 21st century, as the micro level computing power undertook a drastic advancement, smart devices which are capable of generating data for smart monitoring were commercially viable at lower costs per processing power. Device management software platforms are developed to collect data from these devices. Recent advancements in the field of machine learning marked the growth of advanced techniques to analyze the big data generated by smart devices. Analyzed data can be utilized to model implicit building behavior to capture building specific nuances, such as R-factor (insulation level), etc., based on data-driven approaches. These predicted models are accurate than any theoretically built model. In addition, smart device operation can also be controlled from the learned models to optimize user comfort at lower energy consumption. Large-scale application of such edge analytics architecture with data accumulating sensors proliferated throughout the city, led to the birth of new domain of computing termed Urban Computing. Large-scale application of urban computing by inter-connecting buildings and autonomous vehicles gave birth to the new topology of sensor nodes called the smart city. Platforms are developed which can manage all such smart devices. Building energy management software is one such software which can evolve to support the smart city architecture.

2.1.2 Building Energy Management Software

Conventionally Building Energy Management (BEM) software is a processor-based system, which can be installed in the building to monitor and control integrated building's mechanical and electrical equipment ranging from simple sensing devices to complicated ventilation, fire and security systems. It consists of a software program hosted on top of hardware interface. This software program is ideally configurable to support various communication standards to interface with the mechanical devices connected to its hardware interface. Hence, this setup is no different than industrial SCADA systems, i.e., distributed control system (DCS) [3]. A DCS is a computerized control system in an industrial premise which has autonomous controllers distributed throughout the industry which is controlled by a central supervisory control. Although this topology is replicated by a building energy management software, there is a fundamental difference between industrial SCADA system and building automation system. This difference ranges from the objective of automation control to the type of control equipment. An industrial SCADA is designed to control entire infrastructure whereas a typical building energy management is designed to control and monitor certain traditional power-consuming equipment. With recent advancements, BEM software architecture can incorporate additional smart non-traditional loads such as security camera, solar PV inverter, battery storage, etc., into its system. Such capabilities strengthen the purpose of adapting BEM technology not just for energy management but also to improve the quality of life. Hence, BEM software can act as a common platform which is capable of automating devices to optimal operating points, secure home with surveillance and regulate demand in accordance with the power grid. In addition to such exhaustive features, BEM software can additionally coordinate different device operations and revolutionize control strategies. Some examples would be coordination of illumination sensor with the lighting load dimmer etc.

Additionally, collected historical device data can be crucial to improving device operating lifetime and is handy during reporting, information management, and decision-making. Integrating all equipment into a single workstation allows detailed insights to control devices for better performance. Hence, to summarize BEM software offers following benefits:

1) Real-time monitoring and control
2) Alarm and notification
3) Smart coordinated control
4) Energy saving
5) Ease of use
6) Fault detection
7) Security surveillance
8) Improves quality of life
9) Maintenance scheduling
10) Improve device lifetime
11) Demand response

Although above advantages portray BEM software as consumer friendly and enhances quality of life, BEM software penetration into daily life is deterred by

1) Need for supported smart devices.
2) Need for scalable and maintenance free BEM solution.

2.1.2.1 Need for supported smart devices:

Building energy management software gets its data from sensors and smart devices distributed across the network. These act as the senses for BEM software to get relevant data to monitor and make the smart control. Hence, BEM software entirely depends on the number of smart devices in the network. This necessitates growth of smart devices which are BEM software compatible. Ideally, a smart device must have a built-in memory to store, analyze, process the measured data and must implement networking stack to communicate stored data with the devices in the network or with the BEM software. Hence, for mass development of the smart device, efficient methods must be adapted to streamline communication stack for accessibility and reduce the onboard memory requirement. This necessity of using least memory for its internal processing and being securely accessible from any remote end has been a constraint to many software applications in the past. In order to understand how smart devices can address this, it is essential to understand the requirements of a typical smart device.

2.1.2.1.1 Communication Stack:

For a device to interconnect with any other device, it has to implement networking stack compatible with rest of the network. The networking stack is hence, standardized such that device communication specifications can determine interoperability. Since the networking stack involves interface between hardware, operating system and application software, it is not feasible to have a single standard to dictate the entire communication architecture. Therefore, a layered interconnection standard module is designed where each is functionally abstracted from other layers. For a device to interoperate with another, they just need to have similar corresponding layers on both devices. The blueprint of this layered stack is termed as Open System Interconnection model (OSI model).

The OSI model is a framework model which describes how a complete networking stack adapted by a device looks like. In practice, any device to communicate with other devices must implement a similar combination of layers mentioned in OSI model. Each of the seven layers perform a particular function during network communication and only know enough to interoperate with the next layer. Among the seven layers defined in open source interconnection model, important layers are:

1) Application Layer- This layer defines the mechanisms for exchanging application messages, the messages exchanged include error messages, and the security features used to protect the application messages.

2) Transport Layer- This layer defines the reliability of the communication between connected devices. Majority of devices use either TCP or UDP specifications for this layer. Devices which are based on TCP have strong communication reliability since it has frame checking, three-way handshake, etc., whereas devices which are based on UDP specifications are fast due to less overhead but are not reliable.

3) Network Layer: This layer defines the rules to packet forwarding from one network to another. Hence, it largely relies on the way devices in the network is addressed. Predominantly any device which is connected to the Internet uses IP4 addressing as the network layer. In this case, routers implement network layer.

4) Data Link Layer- This is the interface between device software and the hardware communication module. Hence, it defines communication with devices that share physical communications medium. In general, devices use MAC addresses to communicate with each other within the same network.

5) Physical Layer- This is the hardware section of the communication stack which deals with electrical signals/light or radio waves that transmit information from the device to device. Hence, this layer defines

standards for physical voltages, frequencies, and other physical properties. Although communication can be done with just physical layer without any of the upper software layers, it can require a dedicated connection. Hence, software layers' definitions ensure distinction thereby allowing shared bandwidth.

Any device which needs to communicate with another device should implement either a part of or all layers. Although the device can implement any combination of specifications at each layer, the interconnected device should be implementing the same specification for each corresponding layer to make a meaningful communication among each other. Hence, taking this standardization into consideration, early 1990's witnessed a tremendous increase in connectivity of devices by maintaining a standardized protocol termed "Internet" on the Network layer. Figure 1 shows how protocol layers interact between two services.

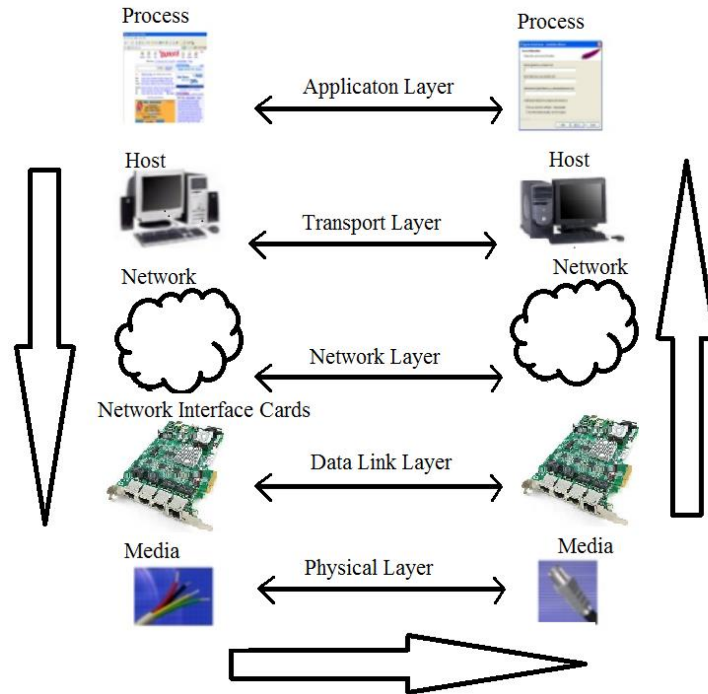

Figure 1: Communication between two services in terms of OSI model

The Internet is a system of interconnection between devices adhering to IP addressing and corresponds to the network layer. Any device which implements Internet protocol suite for addressing itself, then it can be connected to peer IP device through IP network. Hence, the concept of the Internet can be summarized to an implementation of IP standard at the network layer to form an interconnected network. It was initially developed in the 1960s to serve as robust, fault-tolerant communication between computer networks in research-oriented research projects commissioned by the United States federal government. Since then from the mid-1990s, it grew rapidly with over 100 times in magnitude over two decades' time [4]. This widespread adoption led to the domination of Internet as a global system of interconnecting devices.

Traditionally, the Internet has been the backbone of the worldwide web, which was restricted to sharing hypertext resource over the remote location. Due to widespread adoption of the Internet, any application can leverage on existing Internet infrastructure to share its resource to anywhere on the globe. This led to the growth of Internet Protocol supported web applications. Applications of any context take advantage of the abstracted architecture of the communication stack to implement their own standards at application layer and leverage over Internet connectivity by implementing Internet Protocol suite at the network layer. By such implementation, services of different application can form the same network with other applications. This flexibility at application layer implementation has paved way to connect millions of

heterogeneous systems on a common platform called the Internet. Hence, Internet solved the problem of developing an interoperable network of connected devices.

Past few years have witnessed a hike in smart IoT products which adapt to the Internet infrastructure. Forecasting the rate of penetration of smart devices, it is analyzed that by 2020, IoT will comprise up to 26 billion interconnected devices a 30-fold increase from 2009 [5]. In order to facilitate this mass production of smart devices, smart devices had to support plug and play to ease the deployment. Today, most smart device vendors base their device deployment on IP protocol suite to ensure the simplest form of plug and play. This progression towards Internet infrastructure compatible architecture led to a new domain of connected things termed "Internet of Things" (IoT).

2.1.2.1.2 Inbuilt memory requirement:

Any sensing or measuring device needs to maintain memory to process the sensed/measured data. Most devices additionally require storing the processed data. Since IoT devices are developed to operate robustly in any environment, the concept of data processing and storing does not reconcile with its purpose. Initially, this was naively addressed by having devices with sufficient memory on its hardware. This approach increased the device size and had reliability concerns. Thus, this solution was deemed to be inappropriate for any kind of commercial deployment. Realizing the potential in IoT market, a number of cloud software platforms aroused to address this issue by providing hand in hand support to deploy smart devices on the Internet platform with ease and have unlimited memory allocated at the cloud which devices can make use of. These platforms are termed as IoT platforms. The devices can simply push its collected data to the cloud platform where data processing logic is applied. Additionally, these platforms provide services to manage the device and its functionalities from anywhere in the globe. To ascertain benefits of cloud IoT platforms and cloud computing, a deeper look into the cloud technology is required.

2.1.2.2 Scalable and maintenance free BEM solution:

For accurate analysis and smart control, BEM software requires high-resolution data collected from sensors located in different thermal zones to improve the accuracy and model predictability. Since different sensors can adhere to different protocols, such as BACnet, Modbus, RESTful API, etc., BEM software should scale itself to concurrently gather data from the device of any protocol. Hence, BEM software requires being adaptive and scalable to support its functionality irrespective of the number of devices in the network.

Scalability feature dictates the robustness of any software and depends on the physical hardware resources allocated for its computation. A hardware constrained BEM solution can never address scalability feature and depends on RAM and processor capabilities to determine the number of devices it can concurrently support. Hence, software architecture adapted for BEM software must take following characteristics into consideration–

1) Scalability
2) Protocol agnostic(Modular)
3) Secure
4) Easy to use
5) Maintenance free

The problem of being hardware constraint is not only confined to BEM software but also limits any software-defined systems. Hence, similar to smart devices, conventional BEM software can also utilize cloud infrastructure to perform its computing and data gathering. By harnessing such cloud resources, BEM software can perform as conventional BEM software but with unlimited hardware resources.

2.1.3 Cloud Technology

As observed cloud technology was capable of addressing both the downsides of conventional BEM software. Firstly, cloud solution promotes easy deployment of smart devices which can operate at least onboard processing power and can be remotely accessible. Secondly, cloud platform provides the needed scalability and maintenance-free environment to deploy a BEM software which can support multiple buildings, thereby, promoting smart cities. In order to evaluate all the potential benefits from cloud infrastructure, it is important to understand the concept of cloud and its features on which BEM software can be based [6].

The cloud computing is a fancy term used to denote the scenario where computing occurs at a remote location than the location where data is collected. Cloud computing is Internet-based distributed computing architecture that provides a platform for processors to share processing resources. There are a number of cloud hosting vendors who provide support to host developed software in either privately owned or public data centers. Additionally, cloud platforms provide support to store and process data irrespective of the size of data. Software deployed on cloud relies on sharing resources with other deployed software to achieve coherence and economy of scale. With growing number of software applications, cloud computing has been a platform which was introduced to aid commercial deployment a software solution.

2.1.3.1 The Concept of Cloud:

The goal of cloud computing is to develop a collective, well-maintained, resource scalable and easily deployable solution which would ease the deployment time for applications, thereby giving the flexibility for developers to focus only on their application. For concurrently running different enterprise applications on a single cloud instance, cloud providers leverage on virtualization technology. Virtualization segregates the disk memory into many virtual devices so that each virtualized server can co-exist with others and run their own application. Operating System virtualization makes the best use of computing resources and solves platform dependency. Additionally, memory allocation to the virtualized application can be dynamically allocated on demand, thereby, featuring scalable solution for the software. Hence, it is evident that cloud solution is advantageous for growing start-up companies since the reliability and quality of service of the developed software are taken care by the cloud hosting service. Deploying a software package is a collective effort of platform support. These can be categorized as the layered stack.

1) Data Center: Consists of the data center where computer storages are physically installed along with ventilation and security features.

2) Computer Network: Deploying interconnectivity between all the computers such that they can operate as a network.

3) Servers: A highly elastic and reliable server which can run 24*7 with load balancing features.

4) Virtualization: This defines and provides separate disk partition within a computer to dynamically allocate resources to the deployed software.

5) Operating System, Middleware, and Run-Time Environment: Any software has its dependencies defined by runtime environment and operating system used. Due to this appropriate dependency need to be installed on the virtualized disk.

6) Application and Data: This refers to user-defined code which manages requests received from the server. Different applications may share all the above-mentioned features but differ in this layer based on their application.

2.1.3.2 Cloud Services:

Depending on the complexity and application, the software can utilize cloud services up to a certain extent of the stack. Based on this usage, three types of services are defined-

1) Infrastructure as a Service (IaaS): In IaaS, cloud provider takes care of providing the physical infrastructure for deploying software. Cloud provider takes care of up to virtualization. Software developer takes care of the middleware, runtime environment, resource allocation and application for deploying the software. Hence, cloud provider supports entire hardware stack with software infrastructure taken care by the software developer.

2) Platform as a Service (PaaS): In PaaS, in addition to entire hardware infrastructure, cloud provider takes an additional responsibility to provide required scalability features, environment, and dependencies. Software developer focuses entirely on the application. Reliability, resource allocation is handled by the cloud provider.

3) Software as a Service (SaaS): In SaaS, the software developer leverages on standard applications defined by the cloud providers thereby requiring nothing to take care related to cloud infrastructure and can only focus on business strategy and cloud independent software.

2.1.3.3 Cloud Models:

Once the software developer chooses the type of service, various architectures models dictate the way developed software can utilize cloud resources. The apt model is determined by the application and requirement. Common models are

Client-Server model- This is the traditional software architecture where the deployed software acts as a server. Any client can access the server by connecting to the server which resides entirely in the cloud.

Grid computing model: In this model, clusters of independent computers are connected to a supercomputer, which split up a large task among its computing nodes. It can also be referred as distributed computing or parallel computing.

Peer model: This is in contrast to the client-server model where all the computing servers form a distributed system and work without any central coordinating server. Hence, all the components in the system are peers to each other.

Fog computing model: Here, the server is distributed with a part of its services residing close to the client or environment rather than on cloud in order to process data locally and send relevant, useful data upstream to the main server. Since the distributed service processes data, it can be visualized as a component at the network layer in terms of the server as a whole.

dew computing model: This is similar to fog computing architecture, but adds more functionality to the microservice close to the distributed non-cloud environment. In addition, to functioning as a network layer, the non-cloud services also implement application layer code, thereby, decreasing effective server load. This application layer code can also be the front end user interface.

2.1.3.4 Advantages of Cloud Architecture:

Utilizing cloud resources to deploy software has following advantages:

1) Ease of deployment: Since cloud resources have the flexibility to offer up to 100 percent of service, the consumer can deploy his software as early as possible.

2) A wide range of cost options: As the number of cloud-based software's increase, cloud hosting companies have come up with various marketing strategies to increase flexibility in pricing strategies based on usage (unit pricing) and resources utilization.

3) Maintenance free: With dedicated cloud computing resource, there is a dedicated cloud agency which maintains all software stacks built on the infrastructure.

4) Easy access: Hosting services on a public cloud platform assigns a public IP to the cloud services through which deployed software can be addressed and utilized from anywhere on the Internet.

5) Efficient resource usage: Since cloud hosts a pool of applications independently on a common platform, the cost of maintenance and scalability required is comparatively very low than hosting software independently on different localized data centers.

6) Expertise: A software developer with only application-level knowledge can capitalize on cloud provider expertise to develop a successful product.

7) Reliability: With an added advantage of on-demand resource allocation and elastic load balancing, cloud hosting agencies take care of having 100 percent uptime with high degree of scalability

8) Security: To securely deploy software, one has to be highly aware of various strategical intrusions which may affect his software stability and privacy. Relying on cloud hosting agencies, the software developer has the privilege of utilizing a secure platform for his application.

2.1.3.5 Disadvantages of Cloud Architecture:

Although all above factors appear to be a strong proponent for cloud computing, there exist some shortcomings which are to be considered before basing the software on a cloud platform. Few important shortcomings of cloud hosting are-

1) Potential downtime- Since cloud service providers host many clients on a common platform, improper maintenance or resource allocation can cause an outage to all services.

2) Security vulnerability: Since the deployed software is hosted on the cloud platform, the secure nature of the software relays entirely on the cloud framework. A malicious code running in other application can affect all applications and result in cascading failures. Hence, cloud service selected should adhere to state of the art security standards.

3) Privacy: Since the data is stored in the cloud and not on local disks, the privacy of stored data is a question. It is not only vulnerable to other peer applications running on the same infrastructure but also vulnerable to the cloud hosting agency itself.

4) Flexibility: Switching cloud services is not flexible between different cloud vendors. It is not possible to migrate all the existing services. Hence, choosing the right cloud service is essential.

Even though there are above shortcomings, it is evident that wisely used cloud service along with well-designed architecture would deploy the software without any shortcomings. Hence, the cloud computing has gained approval lately evolving more and more cloud-based platforms.

2.1.3.6 IoT Platforms as SaaS:

As mentioned earlier, today a number of cloud providers provide IoT capable software as a service which acts as a platform to connect hundreds of smart sensors and make logical control over them. These are SAAS platforms where IoT platforms serve as the bridge for direct connection among spatially distributed devices and data storage centers. A number of vendors utilize this SaaS platform to generate a RESTful interface to their devices thereby enabling plug-n-play deployment. Once the device connects to the IoT platform, the platform takes care of device accessibility and memory resources. Table 1 displays various software as service IOT platforms offered by various cloud platforms.

Table 1: IOT platforms [7]

IoT Software Platform	Integration	Security	Protocols for data collection
2lemetry - IoT Analytics Platform**	Salesforce, Heroku, ThingWorx APIs	SSL	MQTT, CoAP, STOMP,M3DA
Appcelerator	REST API	(SSL, IPsec, AES-256	MQTT, HTTP
AWS IoT platform	REST API	TLS, Authentication (SigV4, X.509)	MQTT, HTTP1.1
Ericsson Device Connection Platform (DCP)	REST API	(SSL/TSL), Authentication (SIM based)	CoAP
EVRYTHNG - IoT Smart Products Platform	REST API	Link Encryption (SSL)	MQTT,CoAP, WebSockets
IBM IoT Foundation Device Cloud	REST and Real-time APIs	TLS, Authentication (IBM Cloud SSO), Identity management (LDAP)	MQTT, HTTPS
PLAT.ONE - end-to-end IoT and M2M application platform	REST API	SSL, Identity Management (LDAP)	MQTT, SNMP
ThingWorx - MDM IoT Platform	REST API	Standards (ISO 27001), Identity Management (LDAP)	MQTT, AMQP, XMPP, CoAP, DDS, WebSockets

Xively- PaaS enterprise IoT platform	REST API	SSL/TLS	HTTP, HTTPS, Sockets/ Websocket, MQTT

2.1.4 Interconnecting BEM software, IoT and Cloud technology

The revolution in the mass production of smart devices which generate data points for each measured value inundated the data collection barriers. Knowledge generated by these devices is valuable and has a potential to develop reliable enhanced control. Processing of this generated data and supporting concurrent network connections with devices in the network is challenging for any conventional building energy management software. It requires being backed up by high-quality hardware and software framework. Such a requirement was initially solved by having a distributed computing architecture where computation is shared among multiple distributed resources. This again required the physical installation of more than one processor throughout the building, thereby, aggravating the BEM software maintenance concerns. Also, any updates to the configuration of the installed software require physical access to the building infrastructure to upgrade the software. BEM software being central to the connectivity of all devices must be the reliable and efficient system. To ensure robustness of software and along with maintenance free installation, BEM software must not have hardware dependency.

As noticed, smart devices leverage on IoT platforms to address resource availability and ease of use issues. Utilizing similar cloud platform as a service, a BEM software can be hosted on the cloud environment to address its scalability and ease of installation issues. With growing number of cloud-connected devices which push data to the cloud, a cloud-based BEM software can access these devices by authenticating its identity with the cloud platform. A cloud-based BEM software can, therefore, be upgraded to add support to smart cloud devices, just by communicating with the cloud service of the device. Additionally, cloud-based BEM software would not only address its scalability issues but also have cloud advantages. Hence, compared to conventional BEM software, a cloud-based BEM software has following advantages:

1) *Ease of deployment:* Cloud deployed BEM software does not require the installation of any physical hardware at the building vicinity, making BEM software installation non-intrusive and free from any building space requirement. Additionally, by lacking any physical installation, BEM software is free from any tampering due to building activities.

2) *Remote access to BEM software:* By hosting BEM software on Internet/cloud service, a building administrator can access his building inventory from anywhere on the globe by just connecting to the Internet. Thus, cloud-based BEM software with sufficient user-defined roles and authorizations can support a hierarchy of building automation technology from admin to tenants.

3) *Scalability:* Hosting on a cloud platform with unlimited hardware would provide a ceaseless supply of hardware computation resource to form a connectivity platform among a number of devices.

4) *Reliable and Maintenance free:* Availability of the cloud-hosted software is completely taken care by the cloud provider, thereby ensuring reliable availability of the software without any maintenance efforts.

In addition, the future smart grid desires a decentralized network with intelligence distributed across several devices which have to take autonomous decisions, in order to react quickly and efficiently to changes in energy demands, faults, and such events. To implement such a distributed control, traditional BEM software

should be upgraded to act centrally encompassing support to a number of buildings along with deployed urban sensors. This upgraded BEM software would act as a single point of connection among multiple buildings and supersedes conventional hardware constrained BEM software. Such an architecture requires BEM software to be hosted in a scalable cloud environment.

2.1.5 Challenges for Cloud Architecture

Since cloud migration has its impact most recently, only recent smart devices can be based on cloud architecture to be publicly accessible. Most devices even though implement IP stack, are locally addressed and hence reside within local area network. Due to this, cloud-based BEM software cannot access control over the majority of devices which are bound local to the local area network. Hence, apart from addressing cloud-based architecture shortcomings, cloud-based BEM software needs to address its connectivity issue. To understand why cloud-based BEM software cannot connect to the devices in local area network and how to solve this connectivity issue, it is important to understand how Internet protocol works and how various communication protocols operate at the network level. In addition to protocols functionality, a detailed study of how conventional router's limit the devices to local area network is to be understood. Then various techniques to bypass router's screening can be studied to assess the best routing method by which cloud-based BEM software can connect to the devices in local area network.

2.1.6 Concept of IP Addressing and Multi-Networking

Communication between any two devices requires devices to implement compatible networking layers among each other. In order to enable devices of different protocols to co-exist on the same network, devices must have a common base. In order to implement this, all protocol-specific details were abstracted to its application layer. As the application layer takes care of device functionality, devices can now share a common communication protocol as the base at the network layer to address themselves on the network. Internet Protocol is the ubiquitously used network layer protocol defined by almost all protocols to maintain interconnectivity. Any device which has IP protocol at the network layer is addressed with an IP address. Essentially, IP protocol defines addressing methods to label the datagram with source and destination information. Figure 2 displays a typical internet protocol packet which is exchanged between services.

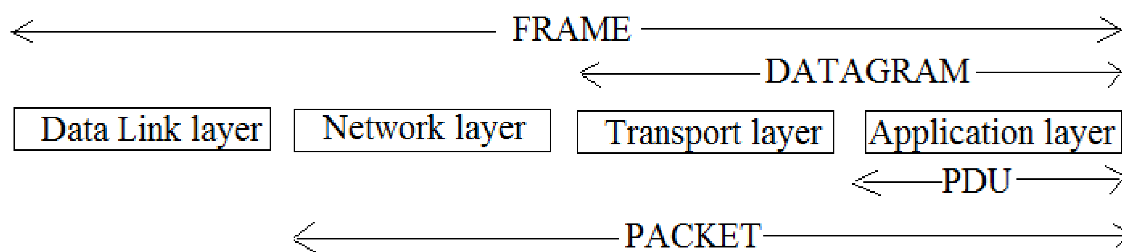

Figure 2: A typical IP packet

An IP packet encapsulates LAYER four packet termed 'datagram' and is transferred from network layer of one device to another device. IP addressing in version four follows a 32-bit number representation scheme in which each device identifies itself uniquely within the Internet network. Due to this exclusive addressing requirement, Internet Protocol version four can have only up to 2^{32} (4,294,967,296) unique addresses in a network. This maximum limit of four billion unique devices appeared to be a shorter number as the Internet grew rapidly. This is termed as **"depletion of IP4 addressing"** and to address this a new IP version six standard was developed. This version is based on 128-bit addressing, thereby theoretically allowing 2^{128} devices to form a network.

Since IP4 addressing scheme had an unrecoverable global reach and lack of interoperability between both versions, the inception of IP6 addressing could not address the depletion of IP4 addressing problem. Hence, in the year 1992, RFC 1335 revision was established which attempted to protect exclusiveness of IP4 addressing by distinguishing IP addresses as private and public addresses based on the address range. Then later distinguished network spaces based on IP addresses. A number of isolated local networks can be established with each network having one device in every network having public addresses and remaining devices sharing a unique private address. The Internet is only the connectivity of devices with public IP addresses, hence making it a network of networks from a simple network. With such an isolated topology, there is no necessity for private addresses to be unique since it is local to its network. Today routers and other Internet providers implement this technique, and it is called IP masquerading. Table 2 shows private IP addresses range.

Table 2: Private IP range

Private IP address range	Number of addresses
10.0.0.0–10.255.255.255	16,777,216
172.16.0.0–172.31.255.255	1,048,576
192.168.0.0–192.168.255.255	65,536

IP masquerading is a technique to hide an entire IP address space, behind a device which is assigned a public IP address along with an internal private address. This device with public IP is called NAT device. The hidden address space consists of private addresses and is addressed to all devices connected to NAT. Any device within private address space communicates with the device on the Internet through the NAT device as a proxy. As a message passes from private IP device, through NAT device, it replaces the message's source address with its own public address and registers the transaction details on its routing table. When the response is received for the sent message, NAT device translates the address to the original address and relays the message back to the original sender.

Figure 3 illustrates a typical local area network topology in a building network.

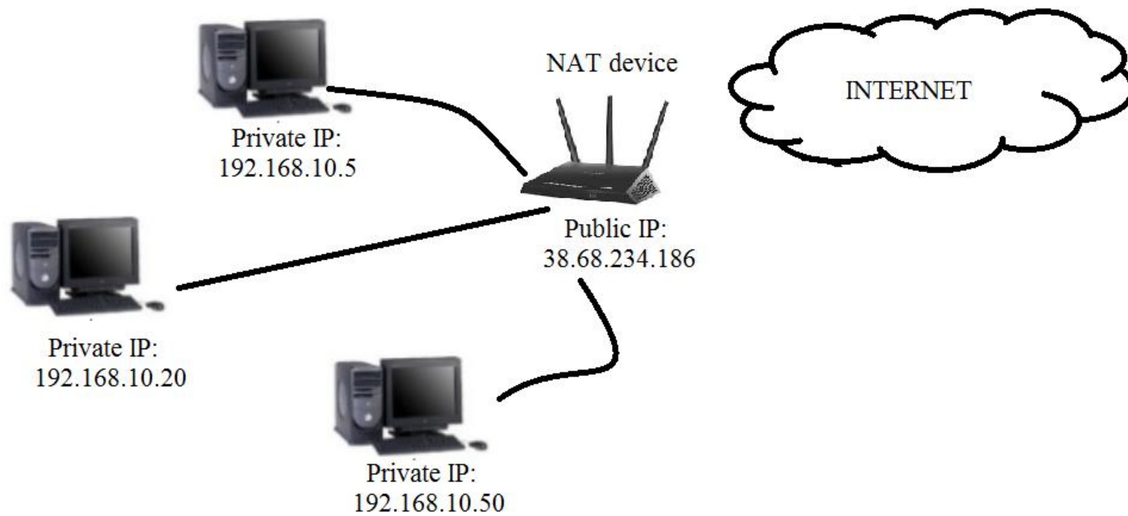

Figure 3: A typical local area network topology

This action by the NAT device is called Network Address Translation and hence the name NAT device. In a residential to the commercial sized building, the traditional router acts as a NAT device. Depending upon the extent of address translation NAT device performs two kinds of translation namely, NAT and PAT. These are distinguished as follows:

NAT (Network Address Translation) uses a pool of public addresses that are mapped one-to-one to the private (or "inside") addresses, keeping the same port number as of internal device. In such scenario, NAT device has more than one public IP address.

PAT (Port Address Translation) uses a single outside public address and maps multiple inside addresses to it using different port numbers. It is called "overloaded NAT or NAPT (Network Address and Port Translation)." This is most common NAT adoption where single outside public address is the address of NAT device. Additionally, there is no standard request/ response mapping defined for NAT device. Table 3 shows different NAT mapping principles, NAT devices follow:

Table 3: Types of Network Address Translation performed by router [8]

	Request Mapping rule	**Response Mapping rule**
Full Cone	Once internal address (iAddr: IPort) is mapped to an external address (eAddr: ePort), packets sent by iAddr : iPort are sent through eAddr : ePort.	Any external host can connect to iAddr:iPort by sending packets to eAddr:ePort.
Address restricted cone	Once internal address (iAddr: IPort) is mapped to an external address (eAddr: ePort), packets sent by iAddr : iPort are sourced as eAddr : ePort.	An external host (hAddr:any) can address iAddr:iPort by sending packets to eAddr:ePort only if hAddr:any received packet from iAddr:iPort.
Port restricted cone	Once internal address (iAddr: IPort) is mapped to an external address (eAddr: ePort), packets sent by iAddr : iPort are sourced as eAddr : ePort.	An external host (hAddr:hPort) can address iAddr:iPort by sending packets to eAddr:ePort only if hAddr:hPort received a packet from iAddr:iPort.
Symmetric	Each time a request is made to a different destination, the router gives a new public address to the same device.	Only an external host that received a packet from an internal host can send a packet back.

2.1.7 Challenges Due to Non-Standardized NAT Architecture

Although the architecture of visualizing the Internet from a simple network to a network of networks had been a fruitful idea to blend in more devices into the connectivity space, this architecture added distance or constraints between the devices in different local networks. Such isolation is beneficial from a security

point of view since devices were secured from directly addressing through the Internet connectivity, it still led to innate problems. In the presence of a NAT device, the internal client is unaware of its own public IP representation. Only the NAT device which rewrites source address and updates the routing table knows about the newly assigned public IP and port of the original sender. Hence, these devices can never be accessible through the Internet since it does not know the assigned public IP. For a device to be addressed in the Internet network, it has to make an outbound request, thereby forcing NAT device to assign a public address for the device. A client on the Internet can never initiate communication with the device knowing its private address. Hence, NAT approach led to a completely different architecture of connected devices where devices are decoupled from Internet network and bind themselves to the local area network. With growing demand for connectivity of things in the domain of telecommunication, gaming, etc., it was required to bind the devices to Internet space to get access to real-time streaming data. Additionally, due to non-standardized translation principles followed by different NAT vendors, a single solution may not address the problem of interconnectivity between private and public network.

Hence, in order to address this issue, certain routers provide a feature to statically update routing table to fix a public IP and port to a service in the local area network. Once, a public IP and port are fixed, any external service on the Internet can communicate with the service in local area network by connecting to the fixed public IP and port. This is called port forwarding. There are many disadvantages due to this design principle:

a) Open ports are vulnerable to any attacker who can scan for open ports by brute force.
b) Not many routers support this.
c) Requires manual configuring each time a new service is added to the local network.
d) Configured public IP must be manually communicated to the client who wants to access the local service.
e) Some routers support only NAT capability due to which ports are not mapped causing only one service to run on given port in private space.
f) Not scalable, maximum services in private space is limited to port availability.
g) Services can't communicate their public address information within data streams which is required for many VoIP, peer-to-peer applications.

Ascribing to above disadvantages, port forwarding is not the best solution to traverse the device in private space to public space. Hence, a number of NAT traversal techniques have been introduced to dynamically map local area service to a public address so as to allow service accessibility to a remote end. Such approaches are called NAT traversal.

2.1.8 NAT Traversal Techniques

NAT traversal techniques are the techniques defined to enable connectivity between devices in the different network by dynamically traversing them to Internet space. There are many such traversing methods introduced with each of it having its own advantages and disadvantages. Most traversing methods require the support of NAT device and hence cannot be a reliable traversal mechanism. Different NAT traversal techniques are: [9]

2.1.8.1 Socket Secure (SOCKS):

SOCKS is a session layer protocol created in the early 1990s to relay communication between client and server through a proxy server when the client cannot directly access the server. This proxy server is called SOCKS proxy. Client software must have native SOCKS support built in, in order to connect through SOCKS proxy. SOCKS client uses a handshake protocol to inform the SOCKS server about the connection

that the client is trying to make, and then SOCKS server connects to the server on its behalf. Famous examples of SOCKS implementation are Tor, Putty, etc.

Disadvantages: - SOCKS requires configuration on each client and is not transparent to applications. Requires a proxy server which adds a lot of overhead. Additionally, it requires router support.

2.1.8.2 Session Traversal Utilities for NAT (STUN):

STUN was initially an acronym for Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators, but this acronym was later revised to Session Traversal Utilities for NAT. Hence, it was initially designed for UDP as transport layer but was extended to TCP.

STUN has a server running on the public side of the Internet which assists as a tool for devices in private network space discover their public IP. The client operating inside a local network sends a binding request to the STUN server. The STUN server saves the source of the request and responds with a success response that contains the saved public IP and port of the client, as observed from the STUN. This response is generally obfuscated through XOR mapping. Once the client in local area network gets its visible public address, it uses this address to address itself in public Internet space.

Disadvantages: - When router uses symmetric NAT principle for mapping, STUN fails because internal devices are assigned different public addresses for different clients they interact. Additionally, it again requires maintenance of a server.

2.1.8.3 Traversal Using Relays around NAT (TURN):

TURN is a relay protocol designed specifically for NAT traversal to act as a workaround to support any router, even if it adopts symmetric NAT. In this approach, there exists a relay server in public network similar to STUN, but this acts as a proxy server for the private server.

The client operating inside a local network sends a binding request to a TURN server which is on the public Internet. The TURN server maintains a persistent connection with the private service, transferring any request made to it, to the private service. Hence, the private address can share the public address of the TURN server to represent itself. The router in between the private server and TURN server has only one routing entry, i.e., the connection between TURN and private server, due to which even if router follows symmetric NAT, private service can effectively route itself to be accessed by any public service. Turn runs on either UDP or TCP transport layer.

Disadvantages: - Since this topology is no longer peer to peer and any communication is relayed through the TURN server, there is a higher latency compared to STUN approach. Also, it uses higher bandwidth compared to original STUN approach. Additionally, it again requires maintenance of an external server.

2.1.8.4 Interactive Connectivity Establishment (ICE):

ICE is a complete protocol which chooses between STUN and TURN to do NAT traversal by picking the best network route available. Hence, ICE is an attempt to make two services to talk to each other as directly as possible by leveraging the best of STUN or TURN depending on the NAT device action.

TURN approach is advantageous only when the required service is behind a symmetric NAT. The frequency of such a scenario where TURN relay is necessary is difficult to pin down but is estimated to be around 8% [10]. Hence, ICE is introduced which provides a communicating peer may discover and communicate its public IP address to other peers by choosing the best among STUN and TURN.

Disadvantages: - Increased complexity in the network architecture. Hence, required only when private service has multiple clients. Additionally, it holds all the disadvantages due to TURN and STUN.

2.1.8.5 NAT hole punching:

Hole punching is the most general technique that exploits how NAT is implemented, to allow previously blocked packets through the NAT. This technique is commonly used for establishing a direct connection between two services in which one or both are behind routers that use (NAT). In case both services are behind a NAT device, to punch a hole through NAT, each client connects to a third-party server that temporarily stores external and internal address and port information for each client and then relays each client's information to the other. Using this information each client tries to establish a direct connection. In case only the server service exists behind a NAT device, then a client can connect to the private service, as long as the service behind a router initiates a connection with the client in the beginning.

Disadvantages: - Requires consistent endpoint translation. Hence, in case consistent endpoint does not exist, period NAT keepalive messages must be sent to make sure initial connection is not closed.

2.1.8.6 Internet Gateway Device Protocol (IGDP):

This protocol is supported by some NAT routers in the home or small office settings and works in conjunction with Universal plug-n-play to provide means for NAT traversal. These routers and firewalls support as Internet Gateway Devices, allowing any local UPnP supported device to perform a variety of actions on the router. These actions include retrieving the external IP address of the NAT device, enumerate existing port mappings, and adding or removing port mappings. By adding an entry on port mapping, a UPnP controller behind the IGD can traverse through the IGD for any external client to communicate with it.

Disadvantages: - As observed success of this traversal depends on router compatibility to support UPnP and its configuration to enable UPnP. In case the router does not support UPnP then, local UPnP controller cannot traverse through the NAT.

2.1.8.7 NAT-PMP and PCP:

NAT-PMP stands for Port Mapping Protocol. It is a network protocol introduced by Apple as an alternative to IGDP for automatically establishing NAT settings and port forwarding configurations without user effort. **Port Control Protocol (PCP)** is a successor of NAT-PMP.

Disadvantages: - Similar to IGDP these are again router dependent and not a reliable approach.

2.1.8.8 Application-level gateway (ALG):

Modern NAT devices are equipped with the intelligence to inspect and fix VoIP traffic in its effort to help with the NAT traversal. This feature in the NAT device is called Application Layer Gateway (ALG) intelligence.

Disadvantages: - It is again router support dependent. Additionally, this approach failed in numerous instances to carry interoperability. The routers deep packet inspection has been unreliable.

2.1.9 Choosing APT NAT Traversal

As observed most traversal techniques required support from the router or NAT device in order to traverse through the NAT. Since IoT gateway application is primarily deployed in a network with legacy devices which are behind age-old NAT devices, it is not a reliable approach to develop IoT gateway relying on router's functionality. Hence, any traversal method which involves router/NAT device support can be discarded. On the other hand, some NAT traversal techniques relayed on the additional server. Since, cloud-based BEM software and IoT gateway act as a peer to peer communicating devices, their communication is not affected even if NAT device follows symmetric port mapping. Hence, from all the above NAT traversal techniques, hole punching is the most popular method of NAT traversal in IoT applications.

2.1.10 Non-Cloud and Cloud Protocols

Hole punching is popularly adopted by smart devices to ensure comprehensive connectivity with their cloud IoT platform. This technique requires the device behind the NAT device to make an initial outbound connection. When an outbound connection from a private endpoint passes through NAT device, the NAT device, translates its private address to represent a public endpoint (public IP address and port number). Any traffic between connected devices is directed appropriately by the NAT device until the connection is closed. Once the connection is closed, the new connection can result in different port mapping making the communication unreliable depending on the architecture. Hence, to ensure stable and reliable connection, IoT gateway and cloud-based BEM software must maintain a persistent connection. Such a persistent connection results in consistent endpoint translations by re-using the same public endpoint. IoT platforms required communication protocols which can conform to such hole punching architecture. Hence, based on their inbuilt capability to support NAT traversal, communication protocols can be distinguished as two types, i.e., cloud protocols and non-cloud protocols.

Cloud protocols are the protocols which are compatible with cloud architecture by inherently taking care of NAT traversal to support multi-networking. As cloud platforms grew to offer device management as software as a service, they require devices to perform NAT traversal when behind NAT devices. There exists a number of data transfer protocols of which cloud platforms require to pick protocols which are lightweight and architecturally support NAT-traversal. This gave birth to new lightweight protocols and amendments to existing protocols to establish reliable and efficient data transfer. Each protocol has its own merits over other protocols, and a number of factors play a crucial role in determining the fit protocol for an application. They are consistent with NAT traversal approaches to traverse the device to make it available to the public Internet network.

Once the device connects to its cloud infrastructure, it maintains a persistent connection with the cloud platform. The device periodically pushes data to the cloud by certain cloud protocol and thereby registers an entry in routing table dynamically. By constantly interacting with the cloud platform, the device maintains its shadow on the IoT platform. Hence, once the device is connected to the IoT platform, this platform acts as a TURN server by providing a RESTful interface to the device. Any cloud-based BEM software can make requests to this cloud platform to monitor the current device status. Based on the request method from BEM software and designed architecture, the IoT platform can communicate with the device for any updates or can simply update the device shadow. Hence, in case of smart devices, accessing the device is simplified ascribing to protocol's built-in NAT traversal mechanism.

To access devices adhering non-cloud protocol, an in-depth understanding of how different non-cloud protocols operate in multi-networking. Based on the protocol operation, software must be designed to communicate with the cloud-based BEM software with a cloud protocol and concurrently communicate with devices adhering to non-cloud protocols. Software design support non-cloud communication stacks along with one cloud protocol stack to interface with cloud-based BEM software. Different non-cloud protocols are analyzed in consecutive sections.

2.1.10.1 Non-Cloud Protocols:

During the inception of data communication between devices, the protocols developed were mainly designed to maintain communication on the wire. Hence, the name non-cloud protocols consist of legacy protocols and certain protocols which were developed without pondering over NAT traversal. These protocols are application specific and do not have any defined mechanism to publish the device into Internet space. Hence, the devices adhering these protocols are always behind the NAT device and to communicate to these devices it is required to be on the wire or LAN. Prominent application layer protocols which are non-cloud and used in building space are-

1) Modbus Protocol

2) BACnet Protocol

3) RESTful HyperText Transfer Protocol (RESTful HTTP)

2.1.10.1.1 Modbus protocol:

Modbus is a communication protocol defined to standardize data communication among automation equipment developed by different vendors. Modbus protocol is architected to follow request-response protocol which is implemented as a master-slave analogy. Any device should have three components to support the Modbus protocol:

1) CPU / microprocessor – To process its measured data and write on the memory block

2) Memory – To hold the measured data and encoded data particulars of the device

3) Communication interface - A means to transfer device data as per request

Before the advent of Modbus, communication was based on Distributed Control Systems (DCS) which were the center of data monitoring and control. Such a centralized control system was too expensive solution to implement automation and involves a lot of arduous cabling. Hence, PLCs substituted DCS for localized monitoring of sensors and worked as independent units to fetch data from the sensors. To make such independent PLCs work in conjunction with sensors a standard protocol was to be defined. It was during this period, Modbus standard was defined as a communication protocol between different PLCs and sensors.

<u>How Modbus works:</u> Modbus supports two interfaces as its physical layer, serial and Ethernet cable. With Ethernet as the physical layer, Modbus supports IP network layer and can use TCP/UDP as a transport layer to make the devices accessible on the Internet [11]. With the serial interface, Modbus implementation is called as master-slave interaction and with Ethernet interface, this is known as client-server interaction. At the application level, Modbus uses function codes and slave ID and memory addressing to communicate between the Master to the slave device. Gateways can be used to convert Modbus RTU to Modbus TCP/IP. Table 4 compares Modbus RTU and Ethernet protocols with respect to OSI model.

Table 4: Comparison of two major variants of Modbus protocol w.r.t standard OSI model

STANDARD OSI MODEL	MODBUS RTU	MODBUS Ethernet
Application Layer	Function codes, memory addressing	Function codes, memory addressing
Transport Layer	-	TCP/UDP
Network Layer	-	IP
Data Link Layer	Serial Line Master-Slave	Ethernet Client-Server
Physical Layer	RS-232 / RS-485/RS-422	Ethernet cable

Frame format: A Modbus frame format is the content of the packet which is transferred from master device to slave device or vice versa. It is composed of an Application Data Unit (ADU) which encloses a Protocol Data Unit (PDU). A different variant of Modbus has different variations of the frame format. Figure 4 shows Modbus RTU frame format in terms of ADU and PDU.
ADU = Address + PDU + Error check
PDU = Function code + Data

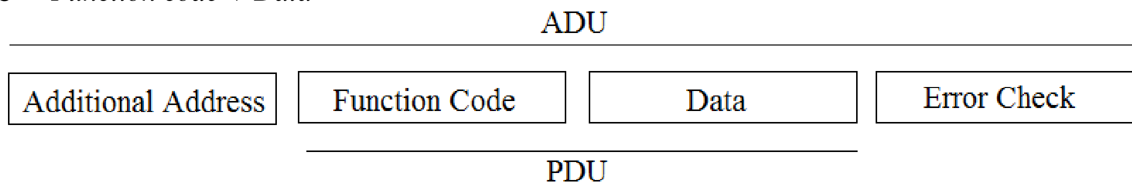

Figure 4: Frame format of a Modbus RTU with ADU and PDU constituents

The MODBUS application protocol determines the format of the request which is made by the client to a Modbus server. This is called Application Data unit and exchange of ADU is called transaction. Function code, a one-byte representation in the request indicates what kind of action, the client wants the server to perform. Valid codes range from 1 to 255 (the range 128 – 255 is reserved and used for exception responses). Additionally, sub-function codes can be added to some function codes to define multiple actions. The data field in the ADU contains additional information that the server needs to perform the action defined by the function code. This can include discrete and register addresses, the number of registers to be handled, and the count of actual data bytes in the field. The data field may also be of zero length in certain kinds of requests. The additional address mentioned in ADU identifies the slave to which command is to be directed. If the request is successfully executed, then the data field of the response contains the data requested. In case of a write request, the data field contains same data elements mentioned in the request. If any error occurs due to any illegal function code or data field, then the data field contains an exception code that the client can use to determine the next action to be taken.

Function codes: A function code is an enumeration value sent by a Modbus master to the slave device to tell which data table it wants to access and what it wants to that table. Data table here is the local memory register of the slave device. There are various function codes to perform an operation different operations on the data table. Each function code performs an operation on a number of contiguous addresses as specified in the data field. There are three types of function codes namely public, user-defined and reserved. The master device specifies a request which has starting address, function code and a number of registers in the data table it wants to operate. Corresponding to the request, slave device responds with a response

answering the request. For example, consider following request and response to reading discrete input coil. Table 5 shows request and response transaction frames between Modbus master and slave device.

Table 5: Request- Response example

REQUEST				RESPONSE		
Description	SIZE	Content(Always in Hex)		Description	SIZE	Content(Always in Hex)
Function code	1 byte	0x02		Function code	1 byte	0X02
Starting address	2 bytes	0x0000 to 0xFFFF		Byte count	1 byte	N
Number of coils	2 bytes	1 to 2000 (0x7D0)		Input Status	N	Values in hexadecimal

Electrical standard employed: Physical layer for the transfer of frame packets in a Modbus network is dependent on the Modbus variant. Serial networks have RS 232, 485, 422 whereas Modbus TCP/IP have Ethernet cable as their physical medium.

RS 232: RS-232 is a technical standard that specifies electrical characteristics of a digital signaling circuit. With this as the physical layer, only two devices can be connected to each other. In such a scenario, one is a master and other is a slave device. Also, with RS 232 the maximum range of data transfer is up to ~15 meters only. Due to these drawbacks, most industries opt RS 485 as the physical medium for their MODBUS RTU protocol.

RS 485: RS 485 is also an electrical standard that specifies electrical characteristics of a digital signal circuit. This termination uses a differential balanced line over twisted pair cable. Through RS 485 medium, the master can connect up to 32 slave devices. With single byte addressing a master can have slaves up to 247 devices, with last eight slave IDs reserved. The slave devices can be spread up to ~1200 meters and are connected to the master in a daisy chain manner in order to avoid collision between communication packages. Figure 5 shows daisy chain connection of slave devices in RS 485 topology.

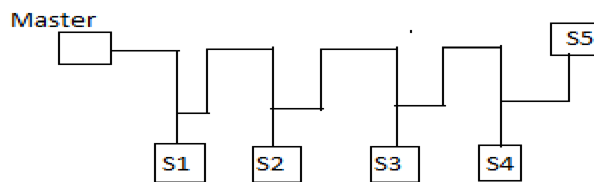

Figure 5: Typical Modbus RS-485 topology

ETHERNET: Modbus serial protocol can be converted to TCP/IP by encapsulating PDU or by adding Modbus Application header (MBAP) as a header to the PDU. Figure 6 shows MBAP header in frame format.

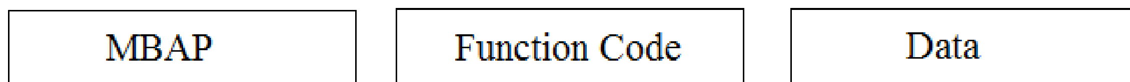

Figure 6: MODBUS TCP/IP with MBAP header frame

Modbus TCP/IP leverages on Ethernet protocol as its physical layer. Ethernet supports MODBUS communication by establishing TCP connection between a Client and a Server. To connect multiple devices of TCP/IP protocol an Ethernet switch can be used.

Multi-networking: Although Modbus started as a serial communication protocol, with the need to facilitate remote accessing features into Modbus protocol various variants of Modbus were developed. For accessing a Modbus device remotely via Internet, it should support Internet protocol. Since serial port cannot support IP network layer, Ethernet is used as physical medium thereby converting Modbus to Modbus TCP/IP protocol. In order to leverage both Ethernet and serial functionality, a gateway which converts Modbus RTU to Modbus TCP/IP network can be used. In most cases gateway adds an MBAP(Modbus Application) header to the existing packet or it can encapsulate entire serial packet into a TCP/IP wrapper. The seven-byte MBAP header includes the following fields:

• Transaction/Invocation Identifier (two Bytes): This identification field is used for identifying Modbus message order when several Modbus messages are sent over the same TCP connection.

• Protocol Identifier (two bytes): This field is always zero for Modbus services, and other values are reserved for future extensions.

• Length (two bytes): This field is a byte count of the remaining fields and includes the destination identification and data fields.

• Unit Identifier (one byte): This field is used to identify a remote server located on a non-TCP/IP network (for bridging Ethernet to a serial sub-network). In a typical slave application, the unit ID is ignored and is echoed back in the response.

2.1.10.1.2 BACnet protocol:

BACnet is a protocol defined to standardize data communication among various building automation and control networks. Compared to OSI model, it has its specification from the physical layer to application layer. Hence, BACnet standard includes everything from the type of cable to use to how to form a particular request or command between discrete vendor devices. Vendors adhering to BACnet protocol have the flexibility to choose the level of conformance. BACnet was designed specifically for all building automation applications:

- ✓ HVAC and Lighting control
- ✓ Fire detection and alarm
- ✓ Security and other household interfaces
- ✓ Utility company interface

Hence, BACnet protocol defines standards for almost all of the building automation and control equipment, thereby, maintaining interoperability between best equipment in Building Automation industry. [12]

How BACnet works: In case of BACnet, interoperability is accomplished by having vendors conforming to the usage of an object-oriented approach for representing entire information encoded within the device. The underlying mapping between BACnet defined standard objects, data and processes links within the device is left to the discretion of the vendor. Once data is in the form of standard objects, standard messages or services are defined for carrying out functionalities between different devices. As a transport mechanism for exchanging messages across the BACnet devices, several LAN technologies were incorporated into the BACnet standard, enabling BACnet to be compatible with any network [13]. Also, Internetworking rules which permit the construction of large networks composed of different LAN types are defined. Hence, in

terms of OSI model, this protocol can be divided with object and services model serving as the application layer, Internetworking model serving as network layer and LAN types representing link and physical layer of the BACnet protocol [14]. Figure 7 shows various compatible technologies on which BACnet architecture is based.

<table>
<tr><td colspan="5">BACnet Application Layer</td></tr>
<tr><td colspan="5">BACnet Network Layer</td></tr>
<tr><td colspan="2">ISO 8802-2
IEEE 802.3</td><td>MS/TP</td><td>PTP</td><td rowspan="2">LonTalk</td></tr>
<tr><td>ISO 8802-2
IEEE 802.3</td><td>ARCnet</td><td>EIA-485</td><td>EIA-232</td></tr>
</table>

Figure 7: BACnet architecture in terms of OSI model

BACnet local area network: BACnet protocol is leveraged on five standard LAN technologies, adoption of which depends on price/performance tradeoff. The fastest among all technologies is Ethernet at ten Mbps with 100 Mbps. Next, comes ARCNET at 2.5 Mbps. For devices with lower speed requirements, BACnet defines the MS/TP (master-slave/token-passing) network designed to run at speeds of one Mbps or less over twisted pair wiring. Echelon's proprietary LonTalk network can also be used in various media. BACnet also defines a "point-to-point" protocol called PTP for use over phone lines or hard-wired EIA-232 connections. All of these networks are LANs. By supporting various LAN technologies, BACnet messages can, in principle, be transported to any network technology. Table 6 shows which LAN technologies BACnet protocol supports.

Table 6: Various LANs supported by BACnet protocol

BACnet LAN	Standard	Data Rate	Packet size(bytes)	Cost
Ethernet	ISO/IEC 8802-3	10 to 100 Mbps	1515 bytes	High
ARCNET	ATA/ANSI 878.1	0.156 to 10 Mbps	501	Medium
MS/TP	ANSI/ASHRAE 135-1995	9.6 to 78.4 kbps	501	Low
LonTalk2	N/A	4.8 to 1250 kbps	228	Varied

Multi-networking: A multi-networked configuration is required in order to resolve communication possibility between BACnet devices in two or more BACnet networks. Such a configuration allows messages to be passed between BACnet devices even though they are on different networks. Networks are identified with a unique "network number." This network number, along with a protocol defined identifier assists in deciding how and when messages should be passed between networks. BACnet facilitates

Internetworking in two topologies; one way is using *router topology* and other by communication through the ***Internet.*** [15]

ROUTER: "Routers" are the devices which interconnect the networks to form a multi-network. By interconnecting two or more LANs, routers pass the messages between the LANs. The router is required to selectively forward messages; without which the entire Internetwork would be overloaded. Each packet carries a destination address which is examined by the router to determine the forward path for the packet.

If the destination address is a device within the LAN, then the message is sent directly to that device. Else if the destination device does not reside on the LAN to which the router is connected, the message packet is forwarded to next router, which can deliver the packet to the destination. For efficient packet transfer, it is necessary for the router to determine which neighboring router is close to the destination. This is accomplished by using "routing tables" in each router. Routing tables contain a list of network numbers in the multi-network containing information on how to reach that network.

INTERNET COMMUNICATION MODEL-IP BASED: On the other hand, for BACnet can leverage Internet infrastructure by adhering to "Internet Protocol" or IP. BACnet has two distinct ways by which messages can traverse an IP network. They are IP tunneling and BACnet/IP. [16]

In IP tunneling, if a device 'A' on network one addresses a message to device 'B' on network two, it sends the message to the router on its local network. This router basically acts like a BACnet router but performs additional tasks due to which it can be called as an IoT gateway. It takes the BACnet message and by looking upon its local table decides corresponding destination router address located on the distant network. It then encapsulates the BACnet message in a User Datagram Protocol frame (a second transport layer) and sends it via IP to the next router. Here both networks are connected via a standard IP router to the Internet. When the router on Network two receives the IP message from its peer, it deencapsulates the BACnet message and sends it to its final destination, Device B. The downside of this approach is that each message has two representations during message passing. This increases the latency of the connection. Additionally, routers are required to maintain a table of its peer routers. Due to this, there is a certain amount of manual programming to maintain the tables, is to be done to all routers each time the configuration changes. Hence, to address this issue BACnet/IP is proposed.

BACnet/IP devices adhere to IP protocol and have an assigned private address so that it is in the local area network. This device's IP address serves the same purpose as a device's MAC or physical LAN address in other BACnet networks. Hence, BACnet/IP devices communicate using IP messages instead of BACnet messages, allowing the devices to be easily added to any IP Internetwork. This concept is termed as "BACnet Virtual Link Layer." In this scenario, BACnet/IP devices do not require any routers and can talk with each other directly over the Internet. The downside of this approach is that there is no way to send broadcast messages on IP based networks. Hence, for this BACnet Broadcast Management Device" (BBMD) is used. This BBMD acts like the router in IP tunneling topology and broadcasts messages on its network. Figure 8 and 9 shows how a message is routed from one BACnet device to another BACnet device in a multi-network architecture.

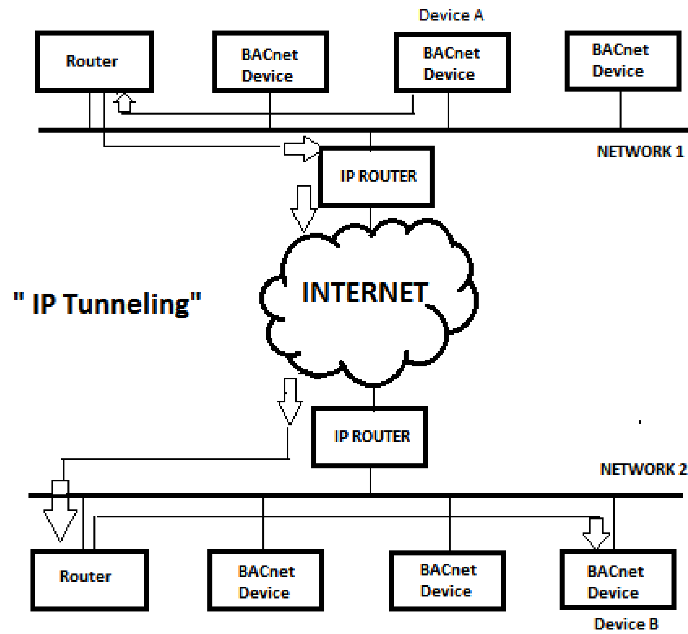

Figure 8: Topology of IP tunneling among BACnet device

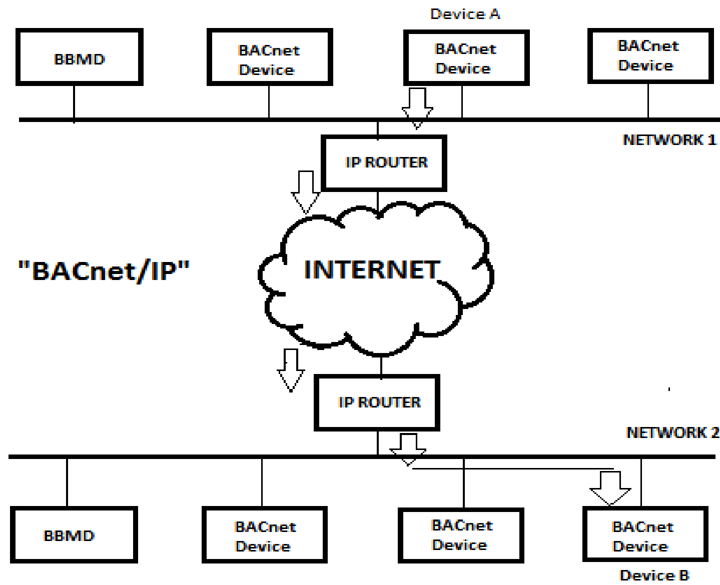

Figure 9: Topology of message transfer among BACnet/IP devices

2.1.10.1.3 HyperText Transfer Protocol:

HTTP stands for Hypertext Transfer Protocol. Hence, in comparison to Modbus and BACnet protocol, HTTP was primarily designed to exchange hypertext information between devices communicating on IP network. Hypertext refers to HTML or web pages. Due to growing popularity of IP devices, HTTP is implemented by certain devices to transfer data in web resource formats such as JavaScript Object Notation (JSON) or Extensible Markup Language (XML).

At the basic level HTTP protocol follows request/response architecture. A client connects to the server and sends a request with the URI, request method, and protocol version, followed by a MIME-like message which contains client information, request modifiers, and body content over a connection. Here, the request methods are generally defined by REST which mentions the type of request whereas the URI points to the resource location. The server responds with a status line that includes the message's protocol version and error or success code. This is followed by a MIME-like message containing server information, entity Meta information, and possible entity-body content. Hence, a typical HTTP message has [17]

1) Message Types
2) Message Headers
3) Message body

Message types: HTTP messages can be categorized as requests from the client to server and responses from server to client. Request and Response messages use the message format defined by RFC 822 for transferring the payload. Both message types consist of a start-line, header fields, an empty line indicating the end of header fields, and possibly a message-body.

Generic-HTTP-message

```
start-line   #Request-Line or Status-Line
*(message-header CRLF)
CRLF      #indicates end of line
[ message-body]
```

Message header: HTTP header fields can be a general-header, request-header, response-header or entity-header fields. Each header field consists of a case-insensitive name followed by a colon (":") and the field value.

Message Headers format

```
message-header = field-name ":" [ field-value]
field-name = token
field-value = * (field-content | LWS)
field-content = <the OCTETs making up the field-value
and consisting of either *TEXT or combinations of token,
separators, and quoted-string>
```

Message body: The message-body of an HTTP message carries the entity-body associated with the request or response.

Message Body

```
message-body = entity-body
| <entity-body encoded as per
Transfer-Encoding>
```

The method type indicates the type of request client wants from the server. HTTP leverages Representational state transfer architectural pattern to define predefined request methods. These are four standard REST methods defined by HTTP protocol:

1) GET- Used when client needs to receive data from the server
2) PUT-Used by client to create or overwrite a resource on the server
3) POST-Used by client to update or modify a resource on the server
4) DELETE-Used by the client to delete a resource on the server

Since RESTful is just architecture design without a standard implementation, it has non-standard payload contents which can be in the form of JSON or XML. The contents depend on manufacturer's implementation and hence can lead to interoperability issues.

Multi-networking: HTTP runs entirely on top of TCP/IP stack. An HTTP server if bound to a local IP runs local to the network and a client from public area network can never reach the server. Such a server can only be assessed by a client running on the local IP network. Hence, a device with an HTTP server needs to be port forwarded or should have a gateway to transfer its data to the remote client.

2.1.10.2 Cloud Protocols:

There are a number of cloud compatible protocols which have a built-in mechanism to push the data to the cloud platform thereby traversing through NAT device. Few protocols are:

1) Advanced Message Queuing Protocol (AMQP)
2) Constrained Application Protocol (CoAP)
3) Web-Sockets
4) Message Queuing Telemetry Transport (MQTT)
5) Representational state transfer (RESTFUL)
6) Data Distribution Service (DDS)
7) Secure Message Queuing Telemetry Transport (SMQTT)
8) Extensible Messaging and Presence (XMPP)

Table 7 shows comparison of various cloud protocol in terms of protocol's architecture, security, Quality of service and transport layer

Table 7: Comparison of protocols which support cloud [18]

Protocol	Transport	QoS	Architecture	Security
MQTT	TCP	Yes	Pub/Sub	TLS/SSL
CoAP	UDP	Yes	Req/Res	DTLS
RESTFUL	HTTP	No	Req/Res	HTTPS
AMQP	TCP	Yes	Pub/Sub	TLS/SSL
Web Socket	TCP	No	Client/Server Pub/Sub	TLS/SSL
DDS	TCP/UDP	Yes	Pub/Sub	TLS/SSL
SMQTT	TCP	Yes	Pub/Sub	Own security
XMPP	TCP	No	Req/Res Pub/Sub	TLS/SSL

2.1.11 Concept and Role of the IoT Gateway

Based on above discussion, it is possible to communicate with devices adhering cloud protocols from a cloud-based BEM software, but devices adhering to non-cloud protocols require a mechanism to traverse through NAT. This traversal mechanism must be augmented on top of every non-cloud device's communication stack to add support for the device to perform NAT traversal. Such an additional layer would act as a network layer for cloud-based BEM software to access devices local to the network.

Additionally, not all manufacturers prefer to implement IP suite. In the past, manufacturers produced a plethora of devices to support their proprietary protocol or implemented communication relying on standard legacy protocols for easing their deployment phase. These are non-IP compliant devices released before the Internet broke out and adapted to the serial interface. Fortunately, due to the necessity for remote monitoring, certain legacy protocols have revised their standards to define IP variants of their protocol. To translate non-IP compliant devices to IP based, a gateway device was designed. This device acts as protocol translator translating IP variant of legacy protocol to the serial communication interface and vice versa. This approach hence traverses non-IP compliant devices to IP space.

Using similar strategy, a similar form of IoT gateway can be introduced which acts as a layer on top of all non-cloud devices to traverse local area devices to Internet space. To communicate to the non-cloud device, a cloud-based BEM software can establish a connection to the IoT gateway and request to the IoT gateway. The IoT gateway makes the appropriate request to the local device. Any communication with the local devices is relayed through the IoT gateway. Hence, this IoT gateway software acts as a super device abstracting all non-cloud devices. Additionally, the IoT gateway must adapt to some traversing technique, to publish itself on Internet space. Since the IoT gateway distributes a part of the computational load on the cloud infrastructure, this model of cloud architecture is called fog computing.

2.2 LITERATURE REVIEW

Internet of things(IoT) is a blooming domain, and a lot of research has been put into it from various aspects, such as cybersecurity, interoperability, etc. In this thesis, a specific issue of a cloud-based device management system is addressed with the development of an IoT gateway. This IoT gateway features two important tasks:

1) Device management
2) Proxy device performing Fog computing

2.2.1 Gateway as device management software

A brief overview of an IoT gateway and its applications in terms of commercial deployment were published in [19]. This paper discussed various message patterns in the IoT gateway context. A number of papers proposed an architecture for such an IoT gateway as a standalone broker device. This device acts as a middleware to perform communication with multiple devices in the network. In [20] authors architect a smart grid BEM solution which can monitor, analyze, and control devices. This building energy efficiency solution is visualized to be one of the critical elements of the future smart grid. Additionally, for communication between devices of discrete protocols, various IoT gateways were proposed. Authors in [21] [22] proposed IoT gateway to integrate different protocols, such as ZigBee, GPRS, Bluetooth, and Ethernet. But the proposed IoT gateways were not flexible and customizable for different applications. In order to address this issue, many multi-protocol scalable integration architectures were developed [23] [24] [25] [26] [27]. With growing need for real-time monitoring of devices, mobile client streaming software's

were developed with a wireless IoT gateway as a hub architecture. In [28] an IoT gateway which can read data from serial port and write data to another serial port was proposed.

All these architectures were not targeted to work in conjunction with cloud-based BEM software.

2.2.2 Gateway as proxy device

With the advent of cloud computing, significant changes took place in the way software architecture is designed. A number of new computational models based on distributed architectures were proposed. These approaches were applied in the development of BEM software as well. In [29], an IoT architecture based on OPC.NET specification was developed. Here, an IoT gateway was proposed which would update an OPC.NET server. This OPC.NET server being installed on the cloud was remotely accessible. The limitation of such an IoT gateway is that it addressed only devices which follow OPC.NET specifications.

This approach where a part of cloud computation occurs at the edge of the network is called Fog computing. This provides opportunities for new applications and services especially for the future of the Internet [30]. In [31] a detailed study was presented which explains how an IoT gateway by fitting in fog computing architecture performs on the basis of Upload Delay, Synchronization Delay, Jitter, Bulk-data Upload Delay, and Bulk-data Synchronization Delay. In [32] IoT gateways as distributed nodes perform machine-to-machine communication for discovery and management of connected vehicles. Authors in [33] [34] [35] [36] [37] proposed how fog computing is more stable architecture than cloud computing.

2.2.3 Commercial IoT gateway products

Today, there are a number of hardware, software and end-to-end vendors who commercially deploy IoT gateways. Companies such as Dell, Cisco, Intel, etc., provide commercial-scale smart IoT gateways which perform dew computing. These IoT gateway products [38] [39] [40] have a wide range of physical input modules from serial connections to Wireless connections. But these IoT gateways require the acquisition of devices specifically made for the proprietary IoT gateway and targets sensor devices. Additionally, these IoT gateway devices are compatible with only the cloud IoT platforms dictated by the vendor.

2.3 RESEARCH GAP

All literature focused on proposing standards and architectural philosophies to integrate either recently developed smart sensor devices or industrial standard specific devices into an IoT platform but not all kinds of devices in building automation domain. Although some literature proposed such platforms, they were industrially targeted requiring devices to conform industrial standards. There was no academic literature on an IoT gateway platform which can work in conjunction with cloud-based BEM software to have access to devices adhering both legacy protocols and modern cloud protocols.

In this thesis, a commercial scale IoT gateway has been designed and developed which targets integration of legacy protocols along with recent RESTful API devices into a cloud-based BEM software architecture. This device acts as a master device polling non-cloud devices in the network and pushes the received data to the cloud-based BEM software through any of the mentioned cloud protocol. Based on the above requirement, the architecture of such an IoT gateway has been developed which is scalable to bring devices of non-cloud protocols into its hood and perform NAT-traversal on their behalf. Such a gateway should not only be reliable and scalable to add support to new non-cloud protocols but should also be secure in terms of data transfer. The major goal of this IoT gateway software is to enhance the performance of a cloud-

based BEM software by locally sharing BEM software functionalities and should prove to be reliable in terms of a local BEM software solution.

Consequently, in Chapter 3, an IoT gateway architecture and design are proposed by implementing it from scratch and reviewing best technologies and practices to choose from at each juncture of IoT gateway development. Later, features of the IoT gateway are illustrated. The developed IoT gateway is built with an utmost focus on reliability and scalability such that the deployment of an IoT gateway along with a cloud-based BEM software outperforms a conventional BEM system. To demonstrate this, in Chapter 4, the IoT gateway performance is tested with respect to its RAM usage.

The implemented IoT gateway has been designed to be scalable both horizontally where it can add support to more communication protocols and vertically where it can add support to more devices. Additionally, the proposed IoT gateway has many extensible features along with the capability to run as a stand-alone system if necessary. Figure 10 displays how a cloud-based BEM software can interface with IoT gateway software.

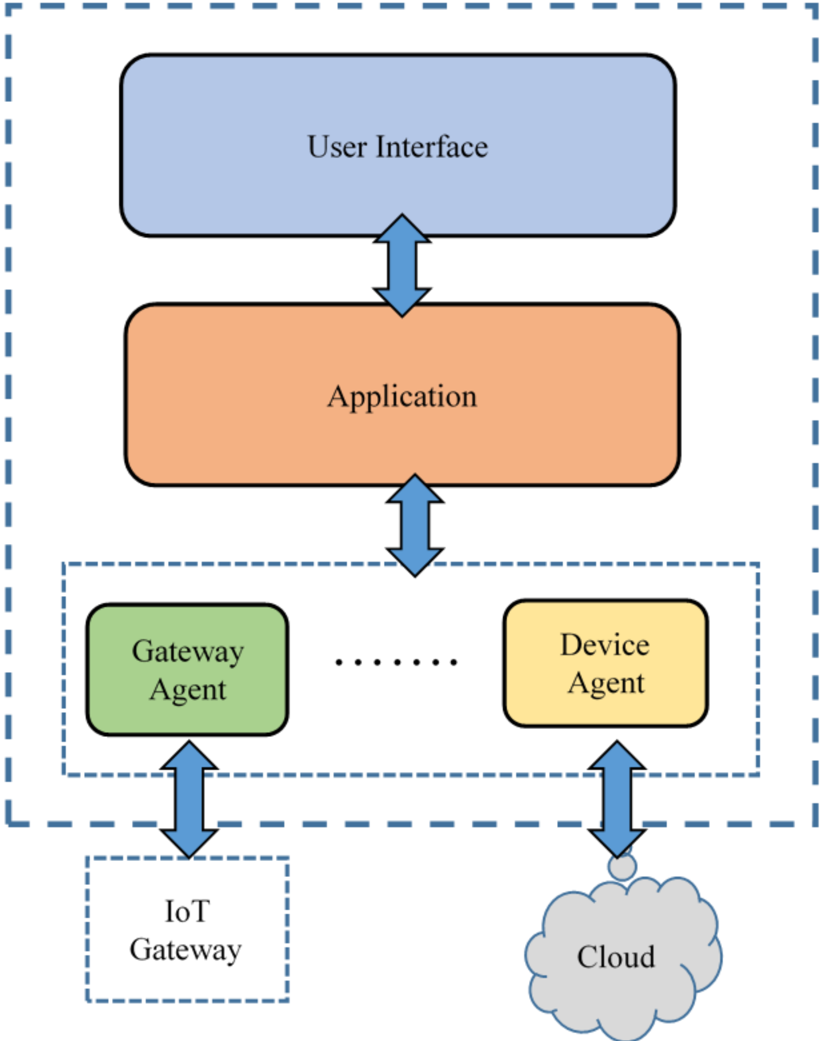

Figure 10: Cloud based BEM software architecture in conjunction with IoT gateway

3. IoT GATEWAY SOFTWARE FEATURES, ARCHITECTURE AND DESIGN

3.1 IoT GATEWAY FEATURES

Gateway software must be architected to discover, monitor and control devices which adhere to non-cloud protocols. Being independently capable of performing device management, it interfaces with cloud-based BEM software to address user's requests. Utilizing such software, any cloud-based BEM software can connect to supported local devices. Hence, the features an IoT gateway needs to support in order to assist a cloud-based BEM software are:

3.1.1 NAT traversal and secure communication with the cloud-based BEM software

IoT gateway software must implement NAT-traversal, to be able to provide comprehensive connectivity to cloud-based BEM software. Once NAT-traversal is performed, a persistent connection is maintained between IoT gateway and cloud-based BEM software.

Communication between these peer devices must be secure. Any communications between peer devices have the following two important security considerations:

1) Authentication: Authentication checking implies the communicating device cross-checks if its connected peer is what it really wants to make a connection with. Since IP spoofing can be done by a third party software to mock as real cloud-based BEM software, such an authentication check protects an IoT gateway from being connected to the malicious controller. Additionally, after authentication, the entire communication between authenticated pairs is encrypted at the socket level thereby securing any sensitive device monitoring data. Also, such an encryption protects from any middle-man attacks which cannot see the transferred data and cannot manipulate the request sent from the cloud-based BEM software to perform undesirable control operations on devices in the network.

2) Authorization: Once peers are authenticated among themselves, an additional security layer is added to the IoT gateway where it checks if the connected upstream has the necessary authorization for any request it makes. This is also called as policy checking where policy refers to the request capabilities the connected cloud-based BEM software possess. In case a cloud-based BEM software fails to have the right authorization code in any of its request messages, the received message is discarded. Extending even more on this, different authentication codes can be set for different types of requests.

IoT gateway software should conform to the above authentication and authorization considerations to ensure communications between the cloud-based BEM software and the IoT gateway is maintained privately to the Internet.

3.1.2 Protocol translators to handle requests

For comprehensive access to a building infrastructure, any BEM software must encompass a wide range of building automation communication protocols. BACnet and Modbus are two such widely adopted legacy protocols which are deeply penetrated into commercial buildings.

Additionally, a large number of next-generation smart devices are deployed basing on the HTTP RESTful architecture. Hence, to ensure comprehensive access to all ranges of building loads, cloud-based BEM software must support these protocols. Since cloud-based BEM software is hosted on WAN; they lack access to these protocols which reside locally on the network. Designed IoT gateway software must be compatible with services provided by the cloud-based BEM software. Hence, IoT gateway software must support BACnet, Modbus and HTTP protocol. Additionally, it should be scalable to support more protocols. Leveraging on protocol stacks, IoT gateway software must be capable of addressing following requests from the cloud-based BEM software:

1) Discover: Should make protocol specific discovery requests to the devices in the network and return a list of discovered devices. Hence, IoT gateway software must be capable of supporting a discovery request.

2) Approve: Once the devices are discovered, the meta-data of discovered devices should be stored temporarily in a lightweight SQLite database with default approval status as pending. Each discovered device must be uniquely represented by a generated "device id" which acts as a primary key to map cloud requests on local devices. To initiate monitoring of devices, a cloud-based BEM software sends an approved devices list represented by "device id's." IoT gateway software should start polling approved devices and relay the collected data to the cloud-based BEM software. Hence, IoT gateway software must be capable of supporting a device approval request.

3) Control: Once the device is being monitored periodically, its current status can be viewed by a user who is operating a cloud-based BEM software. Whenever a user initiates control on the device, a cloud-based BEM software sends a control message along with specific agent id and control parameters. An IoT gateway should make a necessary control action and return Success or failure response. Hence, IoT gateway software must be capable of supporting a control request.

4) Authorize: Certain devices require physical authentication of devices by pressing a button on the device or by any other means. Once physical authentication is done, a device sends a unique ID by which the polling software has authority to monitor or control the device. Hence, IoT gateway software should be capable of supporting a device authorization request.

Figure 11 illustrates the layout of a cloud-based BEM software connecting to an IoT gateway in a local area network.

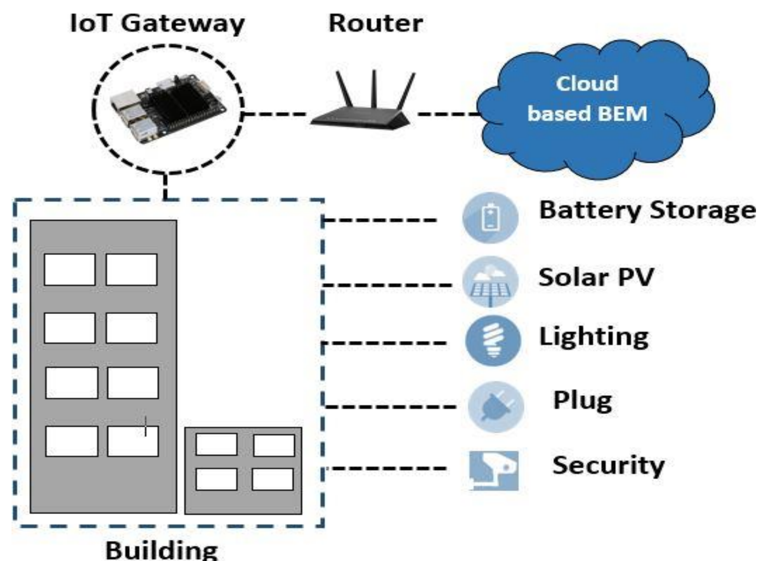

Figure 11: Layout of Cloud based BEM software along with IoT gateway

3.1.3 Software Development Approach

In order to achieve these functionalities, the developed IoT gateway software was based on a multi-processing based concurrent platform. This platform was designed to follow an actor based model, where each process runs independently with a messaging queue as an address box. Each process maintains its own private state and interacts with each other only by passing the message to their address box. In such a platform, a web socket client runs as a process to perform NAT-Traversal and maintains secure persistent communication with a cloud-based BEM software. Once, the client connects to the web socket server running on a cloud-based BEM software; it receives any user request from a web socket server. These requests are addressed by remaining processes which are interfaced with BACnet, Modbus, and RESTful API protocol stacks.

The following section delineates the software architecture and design principles for selecting the above-mentioned approach. The entire software architecture and design are based on Python 3.6.2 as a programming language.

3.2 ARCHITECTURE AND DESIGN

In software engineering, software development has two stages:

a) Software architecture: Architecture of a software defines layout or skeleton of the software. It is the highest level of abstraction of a system.

b) Software design: Software design is about designing individual modules or components. Hence, it defines specific details of each specific method used in the architecture.

3.2.1 Software Architecture

The IoT gateway software architecture must be architected according to the desired functionalities. In a nutshell, an IoT gateway should be capable of receiving requests from cloud-based BEM software and returning appropriate responses. Hence, under its hood, software should provide services to handle requests from the cloud-based BEM software and periodically perform communications with local devices to send back local devices data. This is similar to the conventional Server-Client transaction, where multiple clients connect to the server, and the server handles client's requests. But in case of this application, the IoT gateway requires acting as a client initially by making an outbound persistent connection with the BEM software-based cloud to traverse itself through the NAT into the Internet. Once a persistent connection is maintained, the cloud-based BEM software can then send requests to the IoT gateway. Additionally, messages received by the IoT gateway client should be handled without blocking other incoming messages. To address non-blocking performance during request handling, IoT gateway software must be architected to concurrently run two services at a basic level, with one service handling requests from the cloud-based BEM software and the other service taking care of back-end device transactions.

Concurrently refers to the execution strategy performed during execution of software such that one part of execution does not block execution of another part of software execution. Ideally, parallelism is a much better approach to deal with such unblocked software executions where multiple software executions can occur in parallel depending on the availability of hardware processing core. Since hardware resources are managed by the operating system, any concurrent model built by using operating system calls can achieve parallelism. In case the software runs on a single core hardware, it can never achieve parallelism and can

only achieve concurrency. Concurrency is achieved by scheduling independent code so as to have parts of code executed in a cooperative manner. Hence, software design must take different concurrency paradigms into consideration to develop a code which supports concurrency and parallelism. To support concurrency, software must provide for multiple threads of control. There are three methods by which this can be achieved. They are:

1) Using Multiprocessing
2) Using Multithreading
3) Using Event-Loop

3.2.1.1 Multiprocessing Approach:

A process is an entity provided and managed by the operating system whose purpose is to provide an environment in which program can be executed. Hence, the process provides required memory space for use by the application program, a thread of execution for executing it, and some means for sending messages to and receiving them from other processes. In effect, the process is like a virtual CPU run by the operating system, execution of which would execute an application.

Each process has three possible important states:

- **Blocked** — Where the process is waiting for input.
- **Ready** — Where the process is ready to execute once operating system executes it.
- **Running** — Process is being run on operating system

Each process exists within its own address space, such that no other process can read or write to another one's memory. OS manages entire execution and handling of all the processes. Processes are handled by preemptive multitasking meaning that the OS decides when a process is preempted ("goes to sleep") and which process should go "alive" next. It makes its decision of preemption based on priority and the current state of the process. Just as an application program can leap from current procedure to another by invoking subroutines, the operating system can also transfer control from one process to another on the occurrence of an interrupt or as scheduled or on the completion of a procedure. Since each process maintains its own address space, this “task switching” has considerable overhead each time it occurs. When the operating system switches processes, one thread of execution is temporarily interrupted, and another starts or resumes where it previously left off. Typically scheduling of processes is done based on time of operation of the process.

Processes cannot communicate to each other directly (in modern OS) due to which the OS provides facilities for inter-process communication. Typical ways of inter-process communication involve files, sockets, message queues, shared memory and even memory-mapped files.

Disadvantages with Multiprocessing architecture:

Spawning a process requires creating an entirely new address space. So processes are heavy due to which spawning many of them (in contrast to other concurrency models) is not recommended. Also, the context switching between processes is slow. This requires saving of the entire CPU state (all processor registers that were in use, program counter, etc.) into PCB (usually). Due to these disadvantages in general processes are not scalable and RAM dependent. In general, two GB RAM can support a minimum of 32 processes.

3.2.1.2 Multithreading Approach:

A thread is the smallest sequence of instructions that can be managed independently. Multiple threads can co-exist within one process and execute concurrently while sharing resources such as memory. In particular, the threads in a process share common executable code and the values of its variables at any given time. Due to this sharing, although thread context switching involves restoring of the program counter, CPU registers, and other potential OS data, threading is still less expensive. Threads are therefore called "lightweight processes" as they require fewer resources to manage. The multithreaded environment within a single processor is implemented by time slicing where the central processing unit (CPU) switches between different software threads based on time of execution.

In a multithreaded environment, all threads share same code segment, data segment, and open files. When one thread alters a code segment memory item, all other threads see the changes. Due to this sharing of same address space, threads can conveniently exchange data among themselves without the overhead or complexity of an inter-process communication (IPC).

Disadvantages with Multithreading architecture:

Since threads share the same address space, they may encounter race conditions and other non-intuitive behaviors. Even a simple data structure shared between threads is prone to race conditions if they require more than one CPU execution instruction since two threads may end up attempting to update the data structure at the same time. For data to be manipulated correctly, threads will often need to synchronize to process the data in the correct order. Hence, threads depending on their design might require mutually exclusive operations (often implemented using semaphores) in order to prevent shared data from being simultaneously modified or read while in the process of being modified. Such mutually exclusive approaches when carelessly implemented can lead to another problem called deadlocks. Additionally, since all threads run under the same process. A single thread can crash the entire process.

Apart from threading related issues, few interpreted programming languages have implementations (e.g., Ruby MRI, CPython) that support threading and concurrency, but they can never have parallel execution of threads, due to a global interpreter lock (GIL). GIL is an exclusion lock held by a scripting language interpreter, to prevent simultaneously interpreting the code from more than one thread at once. This limits performance mostly for processor-bound threads, which require the processor, and not much for I/O-bound or network-bound ones.

3.2.1.3 Event Loop Approach:

In above approaches, software architecture relied on the kernel space because operating systems handled concurrent code execution and scheduling. Such an architecture has code execution as per operating systems inbuilt scheduling algorithms, and application software has no control over code execution. This is called preemptive scheduling since threads switch context at moments unanticipated by programmers thereby causing lock convoy, priority inversion, or other side-effects. In order to improve performance and enhance control over concurrency at user space, certain architectures were developed where scheduling of software code occurs at user space, thereby calling it as cooperative scheduling. In this approach, threads relinquish control of execution to other threads voluntarily, thus ensuring code execution is as desired. As a result of such userspace scheduling, context switching within the same process is extremely efficient as it does not require any interaction with the kernel. A context switch can be simply performed by locally saving the CPU registers used by the currently executing user thread and then loading the registers required by the user thread to be executed. This scheduling policy can be more easily tailored to the requirements of the

program's workload. From the point of view of kernel space, the application runs as a single-threaded process since at any given time only one task in the schedule executes. Following are prominently used cooperatively scheduling strategies which can be implemented in python:

1) Callback functions- Procedures are registered for the certain event. When the event is triggered, callback functions are called.

2) Generator functions- Procedures exclusively yield when they encounter yield statement. This approach stores the current state of the procedure execution, so that execution continues from the yield statement.

3) Async/await- This is newly introduced Python 3 feature where a procedure all procedures run in an event loop with each procedure yielding control to the event loop whenever they encounter a blocking operation.

4) Greenlets (Green threads) – This is userspace threads defined by the application developer. These are extremely lightweight and are visualized as a single thread from the operating system point of view.

The basic idea of all above-mentioned approaches is that the software application with different procedures which are needed to be executed concurrently are defined as tasks. These tasks are initially added into a loop called event loop. The software starts executing the first task in the loop. As the executing task encounters a blocking operation, it registers an event on the loop and yields control to the loop. Loop executes next task and checks for any message on registered event whenever it changes context between tasks. As the registered event has some message, the event loop returns context to the older task.

Disadvantages with User Space co-operative scheduling architecture:

Major disadvantage is context switching is dependent on exclusive yielding by a task due to which other tasks can starve if an improperly configured task does not yield. Also, since the operating system takes care of hardware resource allocation for parallel operation, this approach cannot make use of multiple cores thereby limiting the desired concurrency and runs as a single thread from the operating point of view.

3.2.1.4 Comparison

From the above theory, it can be concluded that each concurrent architecture has its own merits and demerits. The right approach depends on the application. Since IoT gateway application runs on a restrained hardware device, it should be architected to use all hardware resources available to it and should implement as fewer processes as possible. Table 8 compares all three concurrency approaches with respect to the features an ideal IoT gateway requires.

Table 8: Comparison of various concurrency approaches

IoT Gateway Requirement	Multiprocessing	Multithreading	Event loop
Efficiently optimize waiting time	Moderately favored	Best favored	Least favored
Uses all CPU cores	Yes	No(Python GIL)	No
Scalable	at least 32 processes for 2 GB RAM	Up to 100’s	Up to thousands
Should work against standard OS blocking calls	Yes	Yes	No, requires monkey patching
No GIL effect	No	Yes	Yes

Operates Without deadlocks and race conditions	Yes	No	No

Based on the above comparison, it can be understood that a properly designed multiprocessing approach would lead to the best architectural design in IoT gateway application. It can make use of dual-core or four-core hardware supplied to it. Additionally, it is robust without any deadlock or race conditions. Since the scalability level is up to 30 processes, the software can be designed to run each process as a communication protocol. With this approach, typically on a 2 GB RAM, IoT gateway can easily support up to 30 communication protocols and a large number of devices adhering to these 30 protocols.

3.2.2 Concurrency Model

In a multiprocessing architecture, each process runs on a schedule determined by the operating system. For communication between these processes, a number of communication strategies are developed. These can be distinguished as synchronous and asynchronous communication based on the concurrency level supported. Synchronous refers to blocking communication between the processes where the sender holds its execution until the receiver responds whereas asynchronous communication implements message passing by which processes communicate without any blocking. Since the processes (Client and Handler) in IoT gateway architecture should exchange messages and also desire the highest level of concurrency, asynchronous message passing is the best technique to implement communication between processes. Additionally, there are a number of other considerations which are required to model concurrent architecture. The design of a concurrent system entails choosing reliable techniques for coordinating process execution, data exchange, memory allocation so that the processes execute in minimize response time and maximize throughput. Hence, a number of formalisms for modeling and handling concurrent programming were developed. One such model which suits IoT gateway application is Actor based model.

3.2.2.1 Actor-Based Model:

The Actor based model was first conceived in MIT Artificial Intelligence labs in the mid-1970's and has recently seen renewed interest ascribing to its decoupled architecture. The Actor based model by itself is pretty simple, but the way it frames the problem creates big advantages in breaking down interprocess communication model into simple, concurrent and fault-tolerant environment. The actor based model adopts the philosophy that "everything is an actor" which is analogous to "everything is an object" notion used by object-oriented programming languages. The entire program's code is divided into different parts, and each part can be run as an Actor individually. Actors do not share their state (i.e., memory) and maintain their own internal state. Actors do not have to spend time managing their environment, establishing and maintaining a communications framework to each other or have any complex concurrency concerns. All of these things are handled by the Actor System which is central to the software architecture. It is the framework on which actors run. Actor System is central to the software performs following tasks:

1) Creates the actual Actor instances.
2) Handles message delivery between Actors.
3) Manages Actor life cycles.
4) Provides appropriate scheduling and concurrency between different Actors.

Due to this centralized actor system, actors are freed to focus on the main concerns of the application. Actor System provides methods for external environment to communicate with internal actors in its framework. Hence, Actor System can easily scale up to run multiple actor systems throughout the distributed network to form a distributed IoT gateway architecture if needed. In a simple IoT gateway application, Actor System can be interfaced with an HTTP web server. While the HTTP web server can be utilized to configure actor

system and other non-application specific operations, the Actor System supports the actual IoT gateway task.

The actor based model operates on asynchronous message passing. Individual processes (actors) send messages asynchronously to each other. Recipients of messages are identified by an address, sometimes called "mailing address." Decoupling the sender from the receiver is the fundamental advantage of the Actor based model enabling asynchronous communication and control structures. Each Actor just hangs out until it gets a message. When it gets a message, it does one of three things: [41]

1. Sends messages to other Actors
2. Creates a finite number of other Actors
3. Updates its internal state

Because Actors do not share state, this asynchronous message sending makes no difference to their implementation. Actor System performs the scheduling of Actor and invokes the receiving Actor's method to handle the received message. Actor message handling has two constraints:

1. Any Actor can only process one message at a time. It must return from the handling of the current message before the next message is delivered to the Actor.

2. Messages sent from one Actor to the other are delivered in the same order that is two different Actors send a message to an actor there is no guarantee about which message is received first.

Actors thus operate on individual messages, running only when a message is received and exiting when they finish handling the message. This message passing architecture is different from the conventional blocking paradigm.

3.2.2.2 Advantages of Actor-Based Model:

The Actor based model is a simple yet powerful tool for the modern software developer. Actor libraries enable the development of applications that are: [42]

• Highly scalable: Actors can be easily created (and destroyed) as needed.

• Fault tolerant: Developed actor based model follows "let it crash" philosophy. The idea is that the program is not coded defensively, trying to anticipate all the possible errors that could happen, but finds a way to handle them because there is no way to think about every single failure point. This philosophy was introduced by Erlang in which every independent actor code has a supervisory actor which takes responsibility to handle crash happenings. Every code run inside a process (Actor) is completely isolated, thereby its state never influences any other process. The supervisor, which can be another actor or actor system, will be notified when the supervised process crashes and it tries to put it in a consistent state again. This makes it possible to create "self-heal" systems. In IoT gateway application, Actor System acting as a supervisor to all actors can notify to the upstream cloud-based BEM software regarding any failures.

• Concurrency support: Each actor has a queue to it, where it can receive multiple messages simultaneously thereby ensuring the quality of service and non-blocking concurrency.

• Loosely coupled and modular: In addition to scalability, it is also naturally promoting a high-modularity. Each Actor can handle a single, specific purpose while other concerns are handled by other Actors. This encapsulated approach results in a loose coupling to ensure modular application design and increase overall reliability and fault tolerance.

• Extensible and adaptive: Since actors are programmed to do specialized tasks, more modules can be added to increase the extensibility of the system. In IoT gateway design, just addition of new actor adds support to the new protocol.

• Location independent: Since actors talk asynchronously using message passing, they can use any mode of communication to exchange message. When socket library is used for such communication, actors can be physically distributed throughout the network, but still, contribute to the overall functionality.

• A Higher level of abstraction: Aspects which are relevant to concurrency such as multi-processing, inter-process communications, scalability, fault tolerance, and recovery, etc., are all handled by the centralized actor System, thereby promoting a friendly environment to build application code.

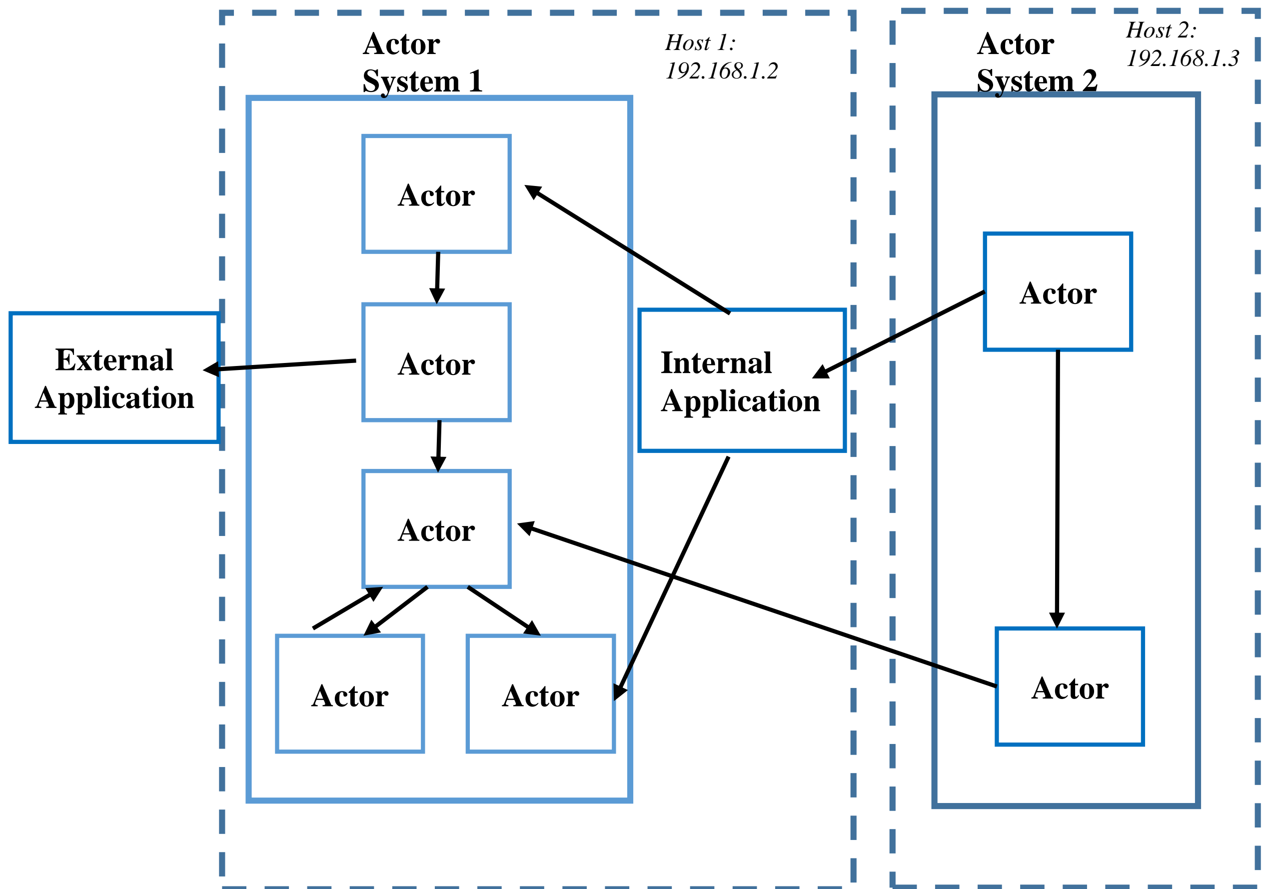

Figure 12: Typical actor based model

Figure 12 shows how a typical actor system is scalable, modular and extensible to develop interaction modules to interconnect communication between different actors located within a single host or in the local area network or in the external Internet. Actors here can represent application code. Thespian a Python library is used to represent the framework. [43]

3.2.3 Software Design

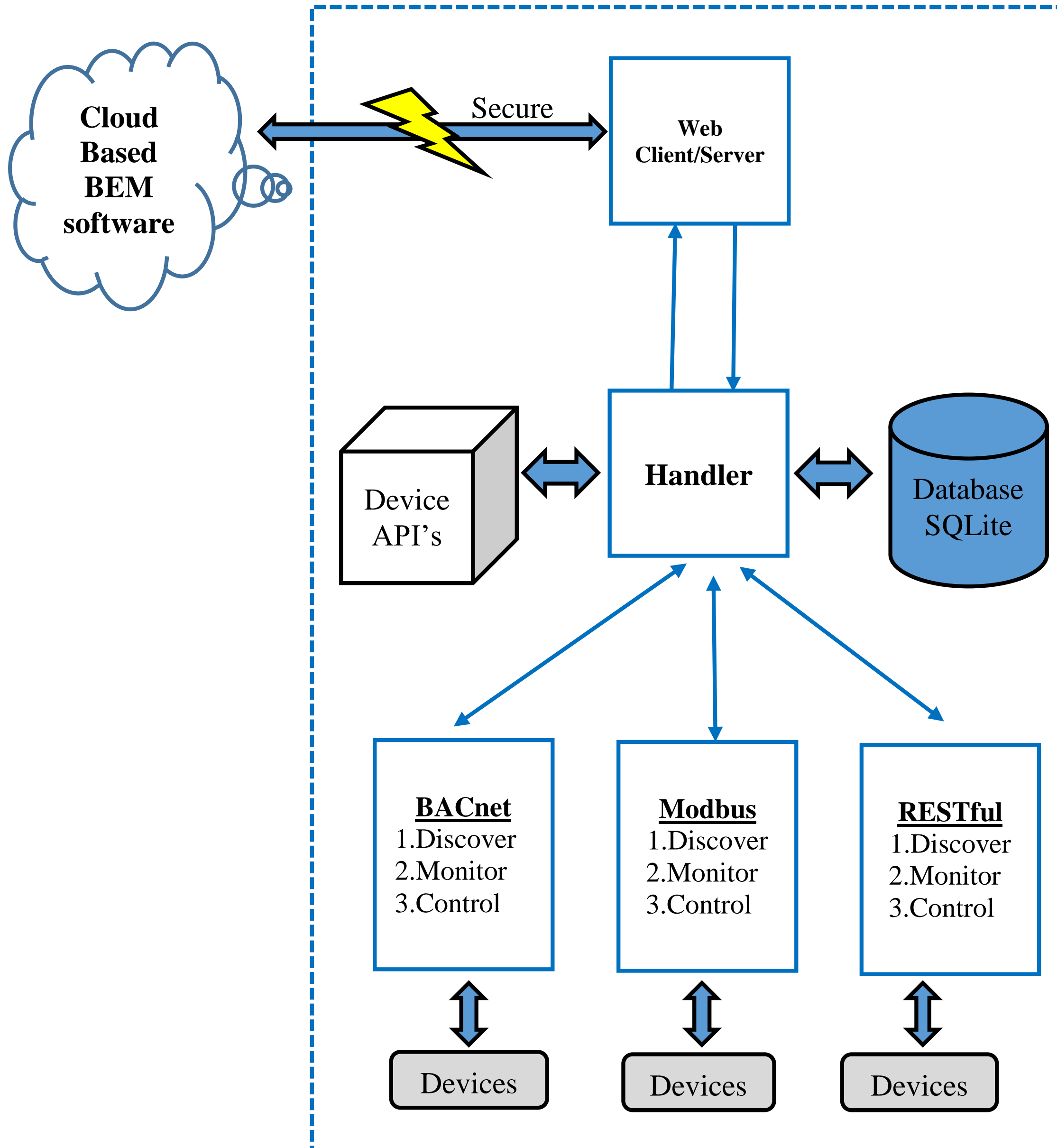

Figure 13: Software Architecture of designed IoT gateway

Figure 13 shows the entire software architecture of the developed IoT gateway. As per requirement, IoT gateway software should be designed to run at least two actors, i.e., Client and Handler actors. The Client actor does the NAT traversal and maintains a persistent secure connection with the cloud-based BEM software. Once it receives a certain request from the cloud-based BEM software, it transfers the request asynchronously to the Handler actor for processing. This design is similar to many web service applications. Hence, software design of IoT gateway software considers the efficient design of two interfaces:

1) Client actor interface
2) Handler actor interface

The design of these interfaces should accommodate basic philosophy on which the IoT gateway is proposed, i.e., integrating non-cloud protocols with a cloud protocol which connects to the cloud-based BEM software. Hence, the design of Client interface involves choosing the best cloud protocol for the IoT gateway application, and the design of Handler interface involves how to efficiently integrate non-cloud protocols to concurrently address the requests from the cloud-based BEM software.

3.2.3.1 Client Interface:

The Client interface along with the Handler interface would be one of the first actor's initiated by Actor System as the software starts. The Client actor must take care of entire cloud connectivity and efficiently transfer requests to the Handler actor. To communicate with the cloud-based BEM software, the Client actor must implement entire communication stack starting from the transport layer, the session layer, and the application layer. It is required to select the apt protocol at each layer from the point of view of IoT gateway software. These three layers work in conjunction to make a secure persistent connection to the cloud-based BEM software, thereby performing NAT traversal.

3.2.3.1.1 Transport layer selection

Transport layer defines protocols responsible for the reliability and state of connectivity between both the peer devices. The protocols adopted at this layer take care of connection-oriented data stream support, reliability, flow control, and multiplexing. These services can be defined as [44]

1) Reliability: A packet sent on Internet network flows through the connectivity path along with millions of other packets. When there is a high influx of packets from connected devices, the sent packet may suffer network congestion and may be lost during transport. By means of an error detection code such as a checksum, the adapted transport protocol can check for corruption of received packets, and thereby acknowledge the sender about received data.

2) Flow control: The rate of data transmission between two peers may be required to be managed in order to maintain a balance between receiving end and processing end. Such a mechanism requires controlling the flow of packets which is taken care by transport layer protocol.

3) Congestion avoidance: As mentioned earlier, a network can be flooded with packets from communicating devices. This situation can be avoided by adding congestion avoidance feature to the flow control to include slow-start. This reduces the bandwidth consumption to a lower level at the beginning of the transmission, or after packet retransmission.

The extent of service support at transport layer depends on the protocol chosen. two types of distinct protocols are predominantly popular in Internet domain, i.e., TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). Both protocols are distinct in terms of performance, and apt transport layer protocol is determined based on application requirements. In order to understand which transport layer suits IoT gateway application, it is important to consider the merits and demerits of TCP and UDP in terms of IoT gateway application.

Transmission Control Protocol (TCP): TCP has built-in flow and congestion control and congestion avoidance. TCP is byte-stream oriented protocol capable of transmitting application messages across multiple packets without any explicit message boundaries within the packets. To achieve this, the connection state is accounted on both ends, and each packet is sequenced and delivered in order. With this connection setup between itself and the remote endpoint, it maintains a consistent connection, thereby, not required to send the endpoint info (IP and port) in each segment it transmits.

TCP protocol establishes a connection in three life phases. The first phase is a handshake process. Once properly connection is made after multi-step handshake process, the connection enters into data transfer phase. After data transmission is completed, it enters connection termination phase when it closes established virtual circuits and releases all allocated resources. Hence, to conclude TCP is preferred transport layer in case application requires making a long-term stable connection.

User datagram protocol (UDP): UDP protocol defines packets to be a self-contained, independent entities of data which carry sufficient information from the source to the destination nodes without any relation with earlier packet exchanges between the nodes. The term "datagram" is reserved for packets delivered via an unreliable service—no delivery guarantees, no failure notifications. Hence, UDP packets are generally referred as datagrams.

The UDP protocol encapsulates user messages into its own packet structure and adds only four additional fields, i.e., source port, destination port, length of the packet, and checksum. Thus, when Internet protocol delivers the packet to the destination host, the host unwraps the UDP packet, identifies the target application by the destination port, and delivers the message to the host application. UDP can be implemented even without the source port and the checksum fields, as they are optional fields. Hence, to conclude UDP is preferred as transport layer when the application requires broadcast or one to many device communications.

Comparison: Based on the above introduction, IoT gateway requirements for the transport layer can be compared with the features offered by TCP and UDP protocols. Table 9 compares TCP and UDP features with respect to the gateway's requirements.

Table 9: Comparison between TCP and UDP

Requirements for IoT gateway	TCP protocol	UDP protocol
Requires point to point connection	Is defined to be point to point connection	Source does not interact with destination
Communication must be reliable	TCP initiates sending packet only after a successful handshake and maintains state	Application payload is directly sent
Secure communication at socket level	Has various security standards built on it ranging from SSL to TLS 1.3	Susceptible to spoofing and DOS attack
Long-term connection	TCP connection based	Not connection based
Ensure Packet flow control between cloud and IoT gateway	TCP protocol has inbuilt flow control mechanism	No inbuilt flow control
Message acknowledgement	TCP supports	No packet delivery guaranteed
Router should record connection longer for efficient NAT traversal	TCP maintains connection state	UDP does not have any connection state

Clearly, the TCP protocol is most preferred over UDP for the IoT gateway application.

NOTE: regarding connection keep alive, although the TCP connection is meant to establish a long-term connection, intermediate NAT-enabled routers may drop the connection after a certain timeout and clear the entry on the router table. Hence, in order to ensure NAT traversal is maintained and the IoT gateway is accessible by the cloud-based BEM software, the IoT gateway application must send connection "keep-alive" messages periodically. These messages are dummy messages which are seen as communication transfer by the router, and the connection is not interrupted.

3.2.3.1.2 Session layer selection

With growing cyber intrusions and cyber attacks, a secure communication path between the IoT gateway and the cloud-based BEM software is of utmost importance. Entire communication between the two peers must be secure and private in order to protect the privacy of transferred data and making sure requests received to the IoT gateway software is from an authentic source, i.e., cloud-based BEM software only. Hence, the communication must include implementing a cryptographic protocol in order to achieve 2 functionalities:

1) Encryption: Hide the data being transferred from any 3rd party intruder.
2) Identification: Making sure the second party receiving the packet is authentic.

These security measures are a part of application and transport layer security. Securing communication can also be added above any layer below transport layer of the OSI model. Hence, there are two encryption schemes:

1) IPSec: securing traffic above network layer is termed as IPsec.
2) SSL/TLS: Securing the packets above transport layer is called secure socket connection.

Internet Protocol security (IPSec): IPSEC security standard encrypts any data above the network layer, due to which it encrypts port and IP addresses of the data. As the packet is transferred through a NAT device, it tries to modify a portion of the packet, specifically the source and destination addresses or ports, when the packet is in route from the given source to the destination. This is not compatible with IPsec purpose which prevents the malicious manipulation of packets between a given source and destination. Hence, IPsec fails to preserve packet integrity and is never a preferred security standard when client/server resides behind a NAT device. To implement IPSec behind NAT, it requires the installation of special routers which can be compatible with IPSec. IPsec is therefore not a preferred security scheme in IoT gateway software.

Secure socket layer/transport layer security (SSL/TLS): SSL/TLS stands for secure socket layer or transport layer security operates above layer four. Hence, it uses an underlying transport medium that provides a bi-directional stream of bytes. [45]

Since it operates above level four, it does not mask the port number and can be categorized to operate at session layer or presentation layer. Over the years, there are a number of cryptographic protocols proposed to address this problem by evolving the originally established standard. These standards are:

1. SSL1 not released publicly
2. SSL2 1994
3. SSL3 1996
4. TLS 1.0 1999
5. TLS 1.1 2006
6. TLS 1.2 2008

TLS 1.3 is still under development and considered to be the even secure than existing TLS 1.2 communication standard. With respect to TLS security means three things:

1. Identity: TLS servers (and sometimes clients) present a certificate, offering proof of who they are so that you know who you are talking to.

2. Confidentiality: Once authenticated, encryption of the connection ensures that the communication remains private between both parties.

3. Integrity: Based on Identity, TLS client can ensure it is connected to authenticate server or else it can simply drop the connection.

The packets exchanged between IoT gateway, and cloud-based BEM software must be understandable to only these two devices. This can only be achieved with encryption.

ENCRYPTION: Encryption and decryption mean to take a source data and transform it into a state which cannot be read by anyone else, and once received at the destination, it is transformed back to its original state by the receiver only. Such an architecture where only the receiver can decrypt requires two elements, i.e., the Cryptographic Algorithm, or cipher, and a key.

A cipher is a mathematical formula that is applied to the original data to encrypt. Only the cipher used to encrypt the data must be used to decrypt at the other end. Due to this in order to ensure compatibility between two ends of communication, cipher algorithm implementation is well known. Standard ciphers such as RC4, Data Encryption Standard (*DES*), Advanced Encryption Standard (AES), etc., have been developed over time.

Since ciphers are deterministic, to add randomness to the communication process, a key is used. This key or series of keys is used along with cipher to encrypt the data in such a way that even receiver requires a valid key to decrypt. Hence, the recipient must know which cipher to use and the key used to decrypt the received data. Based on the type of key used by the recipient encryption can be divided into two types:

1) Symmetric encryption- Both sender and receiver use the same key. Two Internet-connected devices must first share their unique key among themselves for encrypting and later decrypting. Although using the same key has better performance in terms of speed of encrypting the data, it suffers from an innate flaw, i.e., methodology to share the key between two sides. This key can only be sent in clear text before initiation of encryption. This process is hence vulnerable. Hence, asymmetric encryption is proposed.

2) Asymmetric encryption- Both the sender and receiver different key. As the name suggests, this encryption uses two different keys to perform encryption and decryption. These keys are known as the *private* and *public* keys and are mathematically linked to each other. Asymmetric encryption is based on the concept that anything encrypted with receiver's public key can be decrypted by the receiver's private key only. Adhering this principle if sender encrypts data with recipient's public key then only recipient can decrypt the encrypted data, thereby, making sure sent data is targeted to the recipient only. The only disadvantage of this algorithm is that it is computationally more expensive than a shared key cryptography. SSL hence exploits both the asymmetric encryption and symmetric encryption advantages.

ENCRYPTION IN SSL/TLS: Taking advantage of both the encryption algorithms, SSL uses asymmetric encryption during initialization of the connection and then uses symmetric encryption to provide a secure communication channel for the remaining session. In simple terms, first, a client begins a connection to the server requesting for its public key. The server then sends its public key which can be digitally signed by a

certificate authority (CA). Once the client receives the public key, it verifies the authenticity of the public key to confirm the message is from the appropriate server.

Once the server is authenticated, the client creates a random number called Message Authentication Code (MAC). The client uses a pre-shared secret key along with encryption algorithms to randomly generate this MAC. This random number is encrypted by the client using server's public key and sent to the server along with the encrypted version. Only the server with its own private key can decrypt the received message to access the random number. Once decrypted, this random number forms the new shared key for symmetric encryption between the parties. Hence, SSL leverages on the asymmetric encryption to securely transfer randomly generated new private key after which remaining communication is done via symmetric encryption with MAC as the shared key. To explain the entire process briefly,

STEP 1: First, IoT gateway connects to the cloud-based BEM server using TLS. A TLS session is initiated with a handshake.

STEP 2: In the first handshake message, IoT gateway sends the version of SSL/TLS it is using, compression methods it wants to use and cipher suites.

STEP 3: The cloud-based BEM software checks for highest version compatibility and picks cipher suite and compression method. After this basic step, cloud-based BEM software sends its certificate. This certificate acts as a public key for the cloud-based BEM software and also as a certificate for verifying the authenticity of the connecting server.

STEP 4: Gateway looks at the signed public key and checks if server key is authenticated.

STEP 5: Once authenticated, the IoT gateway generates a "symmetric key" (MAC) for use with the Cipher chosen by the cloud-based BEM software. Gateway encrypts this key using the server's public key and sends it back to the cloud-based BEM software. Cloud-based BEM software is the only service which can decrypt this message and to get MAC.

STEP 6: Secure communication using symmetric key cryptography starts from now.

For gateway application, X-509 certificates are generated which are digitally signed with host details. 4096-bit public key and private key are generated for IoT gateway and cloud-based BEM software. Since IoT gateway is a device provided by the cloud-based BEM software provider, they can preinstall all the required certificates to ensure entire communication is secure standard. Also, as noticed, the security of the private key is of utmost importance at both ends, i.e., at BEM software hosted on the cloud and IoT gateway. Since IoT gateway is deployed in the external environment; sufficient care must be taken to make sure private key is not compromised by physically tampering the IoT gateway.

3.1.3.1.3 Application layer selection

This layer deals with the main application protocol of communication between two services, i.e., IoT gateway service and cloud-based BEM software. A number of protocols are designed for this purpose, but the selection of suitable protocol depends on the services and scenario. Application layer protocol can be distinguished based on the architectural pattern. Choosing the right architectural pattern can filter the necessity to survey all protocols. Predominantly there exist two types of architectural patterns by which all protocols can be distinguished. They are:

1) Publish-Subscribe pattern
2) Request-Response pattern

Table 10: Comparison between pub-sub and req-resp pattern

Publish-Subscribe pattern	Request-Response pattern
Prominently used for many to many connectivities	Prominent used for one to one connectivity
Requires additional broker for routing messages	Does not require any broker for routing.
Has a single point of failure(the broker)	Has no single point of failure
Requires additional protocol specifics with hierarchical terms of topics, publish and subscription	No additional protocol-specific terms. Payload can be sent directly.

Table 10 compares publish-subscribe and request-response pattern with respect to architecture's application point of view. Based on above factors, in IoT gateway application, pub-sub architecture would be beneficial if IoT gateways require talking among themselves without the involvement of the cloud-based BEM server. Pub-sub architecture also has a single point of failure, i.e., the broker. Since IoT gateway is designed to be communicating with the cloud-based BEM software as a peer device and maintains a one to one connectivity, request-response architecture suits best as the IoT gateway architecture. Among the cloud protocols surveyed only following protocols support request-response architecture:

1) RESTful HTTP protocol
2) Web Socket
3) Constrained Application Protocol (CoAP)

A connection between two communicating devices is called state. Since cloud-based BEM software requires to connect to the IoT gateway present in the local network area, it needs to retain the initial connection request made by the IoT gateway software. This maintenance of connection ensures long-term connectivity between peer devices and hence is called maintaining state. Table 11 compares various request-response protocols with respect to IoT gateway application. It can be found that Web Socket protocol clearly stands out of remaining two request-response architected protocols and hence can be adopted as a medium of communication between cloud-based BEM software and IoT gateway software.

Table 11: Comparison between CoAP, RESTful HTTP and Web Socket

Gateway requirement	RESTful HTTP Protocol	Web Socket	CoAP
Lightweight	Heavy protocol	Lightweight protocol	Lightweight protocol
Maintains state between communication	Stateless protocol	Maintains state	Stateless protocol
Secure communication	Supports TLS 3	Supports TLS 3	Supports DLS
TCP based	Yes	Yes	No
Long-lived connection	No	Yes	No

Predominantly used for JSON	Predominantly used for hypertext transfer	Predominantly used for JSON, XML data transfer	Predominantly used for JSON, XML data transfer

Web Socket**:** In 2011, Web Socket protocol was standardized, which paved the way to a new low weight, socket kind of flexible protocol. This protocol is currently widely used for transferring data between servers and the browser or direct communication between the browsers. Unlike HTTP, the web socket that is connected to the server stays "open" for communication indefinitely. Through this open connection, data is "pushed" to the client in real-time on demand. Hence, web sockets are low latency and a fully duplex with a single connection.

Web socket communication is initiated by establishing a TCP connection from the client to the server. Once established, the connection is kept open. The initial connection has some additional headers, but this is sent only once per connection, rather than once per request as in HTTP protocol. This drastically cuts down the size of payload transfers between peer communications and saves bandwidth. Later, each payload sent via the socket is framed with only two bytes. The established connection is full duplex, so both the client and server can send and receive data over the same connection. This perfectly fits with the IoT gateway application requirement.

The primary advantages of web sockets in terms of IoT gateway application are:

1. Two-way communication: Cloud-based BEM software acting as web socket server can notify the IoT gateway (client) of anything at any time. Instead of polling the cloud-based BEM software on a regular interval to see if there is something new, the IoT gateway can simply establish a web socket connection and listen to the registered event for any incoming messages on its socket. Whenever the cloud-based BEM software has to send a user request, the server pushes the request to the listening socket. Such type of two-way (full duplex) communication is not possible using plain HTTP.

2. Lower overhead per message: Since IoT gateway software periodically pushes the monitored data from non-cloud devices, it has a potential to generate a lot of traffic between client and server. Due to this scenario, a lower overhead per message protocol would make use of minimal bandwidth. This is because, once TCP connection is established, consecutive messages are sent as a message on an already open socket.

3. Higher Scalability: With lower overhead per message and no client polling, this can lead to added scalability (Single server can handle a high number of clients). Without resorting to cookies and session IDs, the state can be directly stored for a given connection. With this scalability feature available, cloud-based BEM software has can consider an option to run one agent as web socket server for each IoT gateway or one agent as web socket server for many IoT gateways. Ideally, peer to peer architecture with one client-server mapping is more reliable.

Web Socket disadvantages

1. Web Socket is a low-level protocol comparable to a socket on the web. It defines a simple request/response design pattern and hence can't be applied to managing resources.

2. Web Socket lacks many application-level functionalities, such as caching, routing, multiplexing, etc.

3. Not all proxy, DNS, firewalls are aware of Web Socket traffic. Hence, they allow it on port 80 but might restrict Web Socket traffic if snooping is enabled.

3.1.3.2 Handler Interface:

Handler is the back-end of the software which handles any request from the cloud-based BEM software. Since any request is addressed to devices in a local network, the Handler should be having access to all supported non-cloud protocols. Additionally, the Handler should not be blocked during request handling, thus, making itself available for any consecutive request from the Client actor. In this scenario, the Handler interface can take advantage of the actor based model and initiate protocol actors to transfer the request to respective actors. Handler initially uses metadata stored in the lightweight SQLite database to map request to the respective device APIs. Once the Handler maps client requests to device APIs and respective protocol actors, it then dynamically fetches configuration details from device API classes and sends this configuration data asynchronously to appropriate protocol actor. Once Handler transfers the request to the protocol actor, it is ready to handle next request from the client. Protocol actors upon receiving a request message from the Handler invokes its appropriate method to handle the request. These methods talk to the device by making protocol specific communication. To support BACnet, Modbus, and HTTP RESTful protocol, three protocol actors are executed at the back end. These actors are

1) BACnet actor
2) Modbus actor
3) HTTP RESTful actor

3.1.3.2.1 BACnet actor:

This actor is initiated by the Handler once a request from the client addresses any supported BACnet device. Once initiated, BACnet actor runs indefinitely and its lifetime is managed by the Handler actor. This actor provides methods through which cloud-based BEM software can access the supported BACnet device residing in the local area network.

Upon initiation, this actor runs a local BACnet server to communicate with BACnet devices in the local network. BACnet actor supports three important BACnet functionalities. They are:

1) Discover
2) Monitor
3) Control

Discover: BACnet protocol defined a standard method of discovery where a BACnet device in the network when broadcasts WHO-IS message, peer BACnet devices respond with I-AM responses. Hence upon invocation of discover method, BACnet actor creates a virtual BACnet server and broadcasts WHO-IS request in the network. Devices in compliance with BACnet protocol respond with I-AM response. Different types of BACnet devices respond different responses in addition to I-AM response. Depending on which type of BACnet device, it responds following along with I-AM response:

1) If the connected device is BACnet/IP device, it returns IP address as a response along with I-AM response.

2) If the connected device is BACnet MS/TP, it is connected to the multi-network via a router, which broadcasts ‘WHO-IS’ request to all connected devices. These devices respond with I-Am response along

with their device IDs to the router. Router collects these unique Device IDs, attaches its own IP address and sends to the IoT gateway BACnet actor.

BACnet actor collects all such responses and queries for model name and vendor name for each device. Finally, the discovered devices are sent back to Handler actor. Handler actor relays the message back to the cloud-based BEM software.

Monitor: Upon approval of any BACnet device by the user, the cloud-based BEM software sends a request to approve certain BACnet devices, and monitoring of these devices is initiated. During monitoring, BACnet actor starts querying approved devices as per device API and sends periodic messages to the cloud-based BEM software as per configured monitor period. BACnet read multiple properties service message is used to read specific object type's present Value. While collecting monitored data for each device, BACnet actor dynamically maps to respective device API to perform scaling and other conversions on the read data.

Control: Once Handler receives a valid control message on a certain BACnet device, Handler invokes the control method in BACnet actor and transfers all device and control specific parameters. Based on this information, BACnet actor makes a Write Property Request to the appropriate device. If the request is successful, then the device is successfully controlled, and success acknowledgment is sent to the cloud-based BEM software.

As part of source code development, the open source bacpypes library is used [46].

3.1.3.2.2 Modbus actor:

This actor is also initiated by the Handler once a request from the client addresses any supported Modbus device. Once initiated, Modbus actor also runs indefinitely and its lifetime is managed by Handler actor. This actor provides methods through which cloud-based BEM software can access the supported Modbus device residing in the local area network. Upon initiation, this actor runs as a process interfacing to Modbus client library. Modbus actor supports three important Modbus functionalities. They are

1) Discover
2) Monitor
3) Control

Discover: When IoT gateway receives a valid request to discover non-cloud Modbus devices, Discover method of Modbus actor is invoked. Modbus protocol has defined no standard procedure to discover Modbus devices. Hence, it requires polling through all slave ID range to discover slave Modbus devices behind a Modbus TCP IoT gateway.

Hence upon invocation of discover method, Modbus actor looks for any discovery configuration file to generate possible slave IDs list and later performs polling only on those slave IDs. In case there is no configuration file, Modbus actor polls on a slave list between default start and end slave IDs as per configuration. During polling on each slave ID, if Modbus actor receives a response on certain slave ID, it adds that slave as a discovered Modbus device. If the device supports recognition of model name and vendor name, Modbus actor collects its corresponding responses else device identification details are mapped from device configuration file. This response is sent back to Handler actor which relays the message back to cloud the BEM software.

Monitor: Upon approval of any Modbus device by the user, the cloud-based BEM software sends a request to approve certain Modbus devices and start monitoring them. After the first approval message, Modbus

actor starts polling approved devices as per device API and sends periodic messages to the cloud-based BEM software as per configured period. Currently, Modbus actor supports all four read functions. In table 12, function codes corresponding to reading different data tables are represented.

Table 12: Modbus read request function codes

Function Code	Read Register type
1	Read coil
2	Discrete input
3	Holding register
4	Input register

After reading the data, Modbus actor dynamically maps to respective device API to perform scaling and other conversions on the read data.

Control: Once the Handler receives a valid control message on a certain Modbus device, the Handler invokes the control method in Modbus actor and transfers all device and control specific parameters. Based on this information, Modbus device's corresponding registers are written by the Modbus actor. If successfully written, a device returns original request as a response to the Modbus actor, implying write operation was successful. If the request is successful, then the device is successfully controlled, and success acknowledgment is sent to the cloud-based BEM software. Currently, the Modbus actor supports single write register and multiple write registers at a time are given, as shown in Table 13.

Table 13: Modbus write request function codes

Function Code	Write Register type
6	Write register
16	Write registers

As part of source code development, the open source pymodbus library is used [47].

3.1.3.2.3 HTTP RESTful actor:

This actor is initiated by the Handler once a request from the client addresses any supported RESTful device. Once initiated, RESTful actor runs indefinitely with its lifetime being managed by Handler actor. This actor provides methods through which cloud-based BEM software can access the supported RESTful HTTP protocol adhered device residing in the local area network.

Upon initiation, this actor provides an interface to make requests in rest format. RESTful actor supports three important RESTful functionalities. They are

1) Discover
2) Monitor
3) Control

The designed and built IoT gateway supports three of the most popular non-cloud protocols which are widely adopted at building automation level. Each of the above requests is handled by the protocol actors in the following ways

Discover: Most devices supporting RESTful API are discovered by Simple Service Discovery Protocol (SSDP). According to this protocol, when a device in local network multicasts specific M-search method to the multicast address 239.255.255.250 on the port number 1900, then these devices respond via notify method. Since different devices have different M-search message, Handler sends necessary configuration from APIs which details SSDP object message. Once RESTful actor receives addresses of all responding devices, it parses the results and sends the discovered devices list to the cloud-based BEM software.

Monitor: Upon approval of any RESTful API device by the user, the cloud-based BEM software sends a request to approve certain RESTful device and monitoring of these devices is initiated. After the first approval message, RESTful actor starts querying approved devices as per configuration details from device API and sends periodic messages to the cloud-based BEM software as per configured period. Each device API is required to send URL to which RESTful call is to be made. Additionally, it can send data, headers, cookies, files, auth, timeout depending on the vendor. By default, get method is requested which can be changed by sending method parameter. Once the RESTful request is made, the responses are collected in the form of JSON or XML (Since this is not standardized). Hence RESTful actor calls respective Device API class to parse the results dynamically.

Control: Once Handler receives a valid control message on a certain RESTful API device, Handler invokes the control method in RESTful actor and transfers all device and control specific parameters. Based on this information, the RESTful actor makes a Post/get request with control parameters to the appropriate device. The device may respond by 200 HTTP status code or by a success message. According to the device API, response to the control request is parsed. If the request is successful, then the device is successfully controlled, and success acknowledgment is sent to the cloud-based BEM software.

As part of source code development, requests library is used.

By adding more protocol actors, IoT gateway software can provide comprehensive connectivity to supported non-cloud protocol devices.

4. IoT GATEWAY PROTOTYPE DEVELOPMENT AND PERFORMANCE TESTS

4.1 IoT GATEWAY PROTOTYPE DEVELOPMENT

Developed IoT gateway software was designed to work in conjunction with a cloud-based BEM software and performs a part of computation at its vicinity. It addresses requests from the cloud-based BEM by establishing protocol specific communication with the devices in the local area network. Since, device-specific parameters are required to discover a device, parse the discovery message response, monitor a device, scale/parse the monitor response and finally to control the device, protocol actors require interface with device specific parameters coded in device Application Program Interfaces (APIs). Based on protocol specifications, device API is isolated into pluggable modules to the protocol actors. Such pluggable modules generalization depends on protocol relevant parameters. Generalized API modules provide an easy to develop and maintainable platform to add support to new devices. Accordingly, each protocol is analyzed to select standard parameters required from the protocol to talk to the device. The standard parameters deduced from the study are:

1) BACnet protocol is specifically designed for building automation and control due to which it can be considered as the most adopted communication protocol at the building automation level. BACnet protocol defines each data point in terms of Object Type and Index along with its address, and an identifier for every reference point to be monitored. A configuration file approach has been developed where configuration file has above-mentioned protocol relevant details of a given device. API interfaces this configuration file to protocol actors to add support to that device.

2) Modbus protocol, being one of the legacy protocols, has very high adoption in VAV’s, RTU's, sensors, power meters, thermostats, solar inverters, etc. Modbus protocol defines requests to possess Function Code, Start Address, End Address, Dimension, and Multiplier Type for monitoring any of its registers. Hence, a configuration file approach is developed even for Modbus protocol, where the configuration file for a given device has Function Code, Start Address, End Address, Dimension, and Multiplier Type specific details for each readable request. Device API interfaces this configuration file with the Modbus actor to make protocol-specific requests.

3) Along with above-mentioned legacy protocols, RESTful API interface has gained popularity lately due to its simple architecture. Many smart plug loads and other controllers developed during the current IoT era, provide a RESTful API interface to their devices. Since RESTful API is based on HTTP protocol, it binds to local IP and can hence only be accessed by Gateway software in the local network. Any device adhering to the RESTful architecture responds to standard get, post, put methods. Hence, the even configuration file approach can be applied to RESTful devices. But since RESTful is just an architectural pattern and is not standardized, for parsing the responses of the requests made, each device must have an API class.

Although above configuration files assist in supplying and decoding device specific information for devices, still a device-specific Python script is necessary to map device terms to ontology terms. Apart from the device API interface, it is important to develop a fault tolerant framework. For this, the actor must ensure that a fault condition caused by one device must not affect any other device. Hence, to have no interdependency during polling, each actor makes an atomic request to query a device. Any error during data accumulation of polled devices is handled independently in try and exception. Any error or communication failure with one device results in returning a blank dictionary object for that device and actor continues to poll next device. In addition to the simplified device API development, the developed

IoT gateway is designed to collect data even in case of Internet failure. Once the Internet is restored, collected data during the Internet downtime can be pushed to the cloud-based BEM software. Developed gateway stores monitoring device information in a lightweight SQLite database to auto-start device monitoring in case software restarts. Size of currently developed software is 417 KB excluding dependency libraries and Table 14 lists the devices currently supported by the developed IoT gateway.

Table 14: Devices currently supported by IoT-Gateway

Model Name	Vendor Name
Insight	Belkin International Inc.
Socket	Belkin International Inc.
WNC-3Y-208-BN	Continental Control Systems, LLC
Philips hue bridge	Royal Philips Electronics
M1000	Prolon
VC1000	Prolon
LightSwitch	Belkin International Inc.
LMPL-201	WattStopper
On_Off Switch	Vera Control, Ltd.
LMLS-400	WattStopper
LMRC-212	WattStopper

Figure 14 illustrates the laboratory set up of the developed IoT gateway along with various non-cloud devices. The software itself is embedded in an Odroid - a single board computer. Non-cloud devices shown are Wattnode power meter, Wemo Plugload, Philips hue lighting and WattStopper lighting controller.

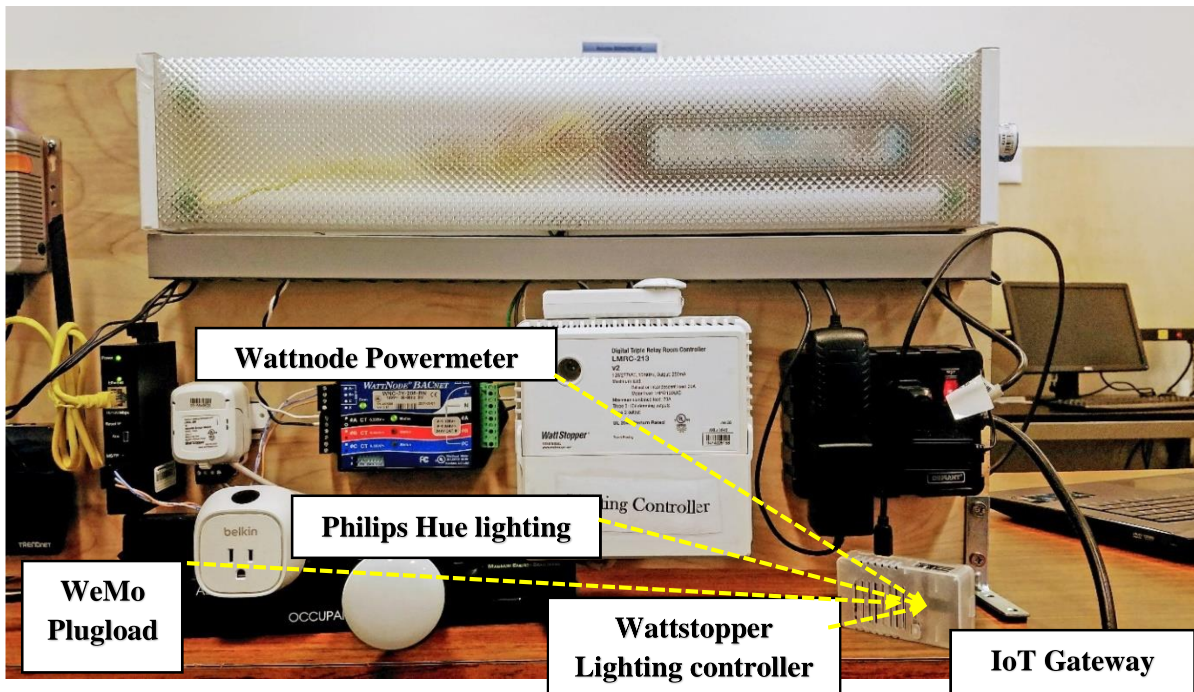

Figure 14: Laboratory setup of the IoT gateway along with few non-cloud devices

4.2 ADDING NEW PROTOCOLS AND DEVICES

Each device API has API info method which mentions the protocol it supports. This data is stored as metadata on SQLite database. Whenever a request is directed to a certain device, based on the metadata, the respective protocol actor is called. Each protocol actor is fed with device API parameters. In case of Modbus and BACnet, these parameters are read from configuration files.

Adding Modbus devices: For adding a Modbus device, a comma separated values (csv) file is required with the method, type, function code, start address, Address, register variable name, dimension and multiplier of each reading point.

Adding BACnet devices: For adding a BACnet device, a comma separated values (csv) file is required with reference point name, BACnet object type and index of each reading point.

The developed platform supports new protocols such as ZigBee, Z-Wave, etc., by adding new actors to the library. New protocol actor must support Discover, Monitor, Control methods. Added new protocol reference must be provided in the handler class.

4.3 PERFORMANCE TESTS

As observed, the design of IoT gateway software was completely based on the best technological choice in terms of scalability and reliability at each stage of development. For evaluating the scalability and reliability of software, various tests can be executed. These tests can be collectively known as performance tests. Software performance testing serves to investigate, measure, validate or verify quality-related attributes of the software. Main attributes are reliability, scalability, and resource usage. Although reliability, scalability, and resource usage are termed differently, they are intrinsically linked to each other. Any software setup running on an unconstrained resource host can be infinitely reliable and scalable. Hence, resource usage can be deemed to be the most basic parameter of which performance of the software can be evaluated.

To understand resource usage, it is necessary to understand why a resource is required in the computing terminology. The entire resource management is dependent on the memory the software requires to run. For software to be executed, it has to be loaded into the processor's RAM for its direct access. This memory allocation is taken care by the operating system. As software requires to deal with more computing operations, it starts to require more space in the RAM. This increase of space in the RAM is inevitable, but the growth of increase in RAM is defined by software designed. An ill-designed software would have a higher rate of the growth of RAM consumption, while a perfectly scalable software would have a low rate of growth in terms of RAM consumption. As the RAM consumption equals to the physical RAM memory, the operating system has to manage the further growth in the RAM usage. Depending on the availability of additional disk memory, the operating system makes a choice to push few of the unused processes to disk memory. This memory swapping would increase the actual RAM memory availability. This concept of memory management is called virtual memory management where virtual memory is the amount of physical RAM along with disk space which is used as memory.

In case the disk space is not sufficient, an interrupt is raised which is listened by the operating system. To handle the interrupt flag, the operating system kills the processes. The killing of the process ensures the process does not crash the entire system. In case operating system does not terminate the processes, the unhandled memory consumption of the processes would eventually crash the entire system. This unexpected process termination raises the question of reliability and stability of the software. Hence, to jot

down the true cause of instability of a software, it is critical to analyze the amount of resources a software requires over time.

In the case of IoT gateway software, it is designed to have three processes, i.e., the Actor System, the Client actor and the Handler actor running and it spawns new processes per protocol. Since the developed IoT gateway supports three protocols currently, on the whole, the maximum number of processes the IoT gateway can run at any given time is six. These are:

1) Actor System- This process acts as the parent which spawns remaining processes. It also takes care of inter-process communication between the spawned child processes.

2) Client actor- This process is the Web Socket client which connects to the cloud-based BEM software and maintains a persistent connection.

3) Handler actor- This process takes care of all request handling. Based on request parameters it spawns more protocol actors and transfers request to these actors. Any responses from protocol actors are transferred to Client actors

4) BACnet actor- This process runs as a BACnet device and addresses any BACnet specific communication.
5) Modbus actor- This process runs as a Modbus master and addresses any Modbus specific communication.

6) RESTful actor- This process makes device specific local HTTP GET/POST requests.

Apart from these six processes, no new process is forked by the software. As a new protocol is integrated, a new process is spawned increasing the maximum processes required by software by one. The commonly used building automation protocols are 1-Wire, BACnet, DALI, DSI, Dynet, EnOcean, KNX, LonTalk, Modbus, X10, ZigBee, INSTEON, xAP. Since the number of commonly used non-cloud protocols are hardly 20; it can be concluded that spawning a process per protocol approach will ensure IoT gateway to spawn a limited number of processes. As the process count is limited in IoT gateway software, the factors affecting scalability is only the memory usage per process.

The process memory usage on a Linux operating system is a complex parameter since it involves virtual memory, transient storage, shared memory, etc. Each process can be identified by a unique number called process Identifier (PID). For analyzing the memory usage by a process, this PID can be used to map to the process and get the current memory consumption details. For this thesis, to measure the memory consumption of IoT gateway software, a python package named psutil is used. With this package following memory usage tests are conducted:

1) IoT gateway software running Idly.
2) IoT gateway software running with one protocol actor and one device.
3) IoT gateway software running with one protocol actor and ten devices.
4) IoT gateway software running with one protocol actor and 40 devices.
5) IoT gateway software running with two protocol actors and ten devices each.
6) IoT gateway software running with two protocol actors and 40 devices each.
7) IoT gateway software running with three protocol actors and ten devices each.
8) IoT gateway software running with three protocol actors and 40 devices each.

In all scenarios, RAM usage is measured in three trials to get the average memory usage. This performance is benchmarked on Ubuntu 16.04 LTS, Intel Core™ i7-4770 CPU @ 3.40GHz with four GB RAM specifications. Since it is not possible to have 120 devices in total, for the sake of performance testing, 15 real devices, and 105 virtual devices are used. These virtual devices run as real devices but generate null as a response to any requests, hence, using virtual devices doesn't affect the overall performance.

4.2.1 Idly running

When IoT gateway software is ideally running, then it has only three processes running in it, i.e., Actor System, Client, Handler actors. Table 15 displays RAM usage of these processes:

Table 15: RAM consumption by software when idle

	Trials	Handler actor(MB)	Client actor(MB)	Actor System(MB)	Total(MB)
Idle Software	Trial 1	34.12	33.23	32.3	99.65
	Trial 2	34.5	32.1	32.54	99.14
	Trial 3	33.1	33.8	33.04	99.94
	Average	**33.9067**	**33.04**	**32.6267**	**99.57667**

The total RAM consumption when software is idle is 100 MB.

4.2.2 IoT gateway software running one protocol actor and one device

When IoT gateway software is monitoring one device, it has four processes running, i.e., the Actor System, the Client, Handler and respective protocol actor.

Table 16 displays RAM usage of these processes:

Table 16: RAM consumption by software when one protocol actor is monitoring one device

	Trials	Protocol actor(MB)	Handler actor(MB)	Client actor(MB)	Actor System(MB)	Total(MB)
With 1 protocol actor monitoring 1 device	Trial 1	32.89	33.1914	33.3	32.75	132.1314
	Trial 2	33.87	35.16	33.69	33.73	136.45
	Trial 3	33.45	34.15	33.992	33.55	135.142
	Average	**33.40333**	**34.16713**	**33.66067**	**33.34333**	**134.5745**

Hence, when software is monitoring only one device, the approximate total RAM consumption is 134.55 MB.

4.2.3 IoT gateway software running one protocol actor and 10 devices

When IoT gateway software is monitoring ten devices of the same protocol, then it still has the same four processes running. Table 17 displays the RAM usage of these processes:

Table 17: RAM consumption by software when 1 protocol actor is monitoring 10 devices

With 1 protocol actor monitoring 1 device	Trials	Protocol actor(MB)	Handler actor(MB)	Client actor(MB)	Actor System(MB)	Total(MB)
	Trial 1	36.65625	37.7265	37.11	35.57	147.0628
	Trial 2	36.4	36.55	37.343	36.23	146.523
	Trial 3	37.02	37.34	37.17	35.69	147.22
	Average	**36.69208**	**37.2055**	**37.20767**	**35.83**	**146.9353**

The approximate total RAM consumption when software is running with 10 devices is 146.93 MB

4.2.4 IoT gateway software running one protocol actor and 40 devices

With the same configuration as above, 40 devices are monitored by a single protocol actor. Table 18 displays RAM usage of software:

Table 18: RAM consumption by software when 40 devices are monitored by 1 protocol actor

With 1 protocol actor running monitoring 40 devices	Trials	Protocol actor(MB)	Handler actor(MB)	Client actor(MB)	Actor System(MB)	Total(MB)
	Trial 1	33.43	34.56	33.23	34.58	135.8
	Trial 2	32.085	32.55	33.34	30.769	128.744
	Trial 3	34.1	32.3	32.34	33.71	132.45
	Average	**33.205**	**33.13667**	**32.97**	**33.01967**	**132.3313**

In this case, approximately 132.33 MB is used.

4.2.5 IoT gateway software running two protocol actors and ten devices each

When IoT gateway software is running two protocol actors, then it has only five processes running, i.e., Actor System, Client and two Handler actors. Table 19 displays the total RAM usage of these processes:

Table 19: RAM consumption by software with two protocol actors and 10 devices each

With 2 protocol actors running monitoring 10 devices	Trials	Protocol actor 1 (MB)	Protocol actor 2 (MB)	Handler actor(MB)	Client actor(MB)	Actor System(MB)	Total(MB)
	Trial 1	34.23	33.65	34.1	33.11	32.98	168.07
	Trial 2	33.45	34.89	33.65	33.34	30.769	166.099
	Trial 3	34.12	32.2	33.23	34.77	32.6	166.92
	Average	**33.93**	**33.58**	**33.66**	**33.74**	**32.12**	**167.03**

In this scenario, approximately 167.03 MB RAM is consumed in total.

4.2.6 IoT gateway software running two protocol actors and 40 devices each

With same configuration each protocol actor monitoring 40 devices RAM usage. With 80 devices, Table 20 displays RAM usage of the software:

Table 20: RAM consumption by software when 80 devices are monitored

With 2 protocol actors running monitoring 40 devices	Trials	Protocol actor 1 (MB)	Protocol actor 2 (MB)	Handler actor(MB)	Client actor(MB)	Actor System(MB)	Total(MB)
	Trial 1	35.56	34.7	33.89	36.1	33.46	173.71
	Trial 2	34.56	32.56	32.43	34.23	33.1	166.88
	Trial 3	34.39	34.12	34.87	33.14	34.12	170.64
	Average	**34.84**	**33.79**	**33.73**	**34.49**	**33.56**	**170.41**

Hence, approximate total RAM consumption even after 80 devices are monitored is 170.41 MB.

4.2.7 IoT gateway software running three protocol actors and ten devices each

In this case, IoT gateway software has six processes running. This is the maximum amount of processes the IoT gateway can spawn. Table 20 displays memory usage as measured while monitoring 30 devices is:

Table 21: RAM consumption by software when 30 devices are monitored with 3 protocol actors

With 3 protocol actors running monitoring 10 devices	Trials	Protocol actor 1 (MB)	Protocol actor 2 (MB)	Protocol actor 3 (MB)	Handler actor (MB)	Client actor (MB)	Actor System(MB)	Total (MB)
	Trial 1	32.45	31.7	33.47	31.89	33.12	33.65	196.28
	Trial 2	34.12	34.65	32.99	34.23	34.56	33.68	204.23
	Trial 3	35.78	36.3	35.83	35.23	35.76	36.3	215.2
	Average	**34.12**	**34.22**	**34.1**	**33.78**	**34.48**	**34.54**	**205.24**

In this scenario, approximately 205.24 MB RAM is consumed by the software.

4.2.8 IoT gateway software running with three protocol actors and 40 devices each

In this scenario, IoT gateway software monitors 120 devices in total with 40 devices per protocol. Table 22 displays RAM consumption by the software when 120 devices are monitored:

Table 22: RAM consumption by software when 120 devices are monitored

With 3 protocol actors running monitoring 40 devices	Trials	Protocol actor 1 (MB)	Protocol actor 2 (MB)	Protocol actor 3 (MB)	Handler actor (MB)	Client actor(MB)	Actor System(MB)	Total(MB)
	Trial 1	32.83	33.42	31.72	32.53	33.4	33.44	197.34
	Trial 2	31.95	32.61	32.64	31.75	32.53	31.57	193.05
	Trial 3	32.64	31.64	33.61	34.42	31.62	32.7	196.63
	Average	**32.47**	**32.56**	**32.66**	**32.90**	**32.52**	**32.57**	**195.67**

This is the extreme scenario compared to all the considered scenario's in which RAM usage is tested. Here it monitors 120 devices with 40 devices per protocol. In this case, the software uses only 196.71 MB RAM.

4.3 PERFORMANCE COMPARISON

As noticed the RAM consumption remains almost constant even with growth in the number of devices monitored. RAM consumption has a step increase for every new protocol. Hence, it can be considered that the RAM usage is independent of the number of devices monitored by IoT gateway software. This makes the developed IoT gateway highly scalable in terms of devices it needs to monitor. This scalability makes the IoT gateway highly fault tolerant in stress tests. Also, since it is noticed that the RAM usage remains almost constant for one device or 40 devices, RAM consumed by the process is allocated by the OS and actual RAM usage is much less than this. Hence, the RAM requirement of the software can be considered further less considering its deployment on a lightweight operating system. Considering 35 MB per process, developed IoT gateway software is found to use a maximum of 250 MB of RAM. As the operating system requires additional RAM to run other functional processes, the IoT gateway can robustly run on a 500 MB RAM constrained hardware. A system with two GB RAM would ideally be more than sufficient to run IoT gateway software robustly. Figure 14 shows the RAM usage versus the number of processes running.

Table 23: Comparing RAM usage in various scenarios

Gateway Status	Total RAM (MB)trail 1	Total RAM (MB) trail 2	Total RAM (MB) trail 3
Idle	99.65	99.14	94.94
1 Protocol actor running with 10 devices	147.0628	146.523	147.22
1 Protocol actor running with 40 devices	135.8	128.744	132.45
2 Protocol actors running with 10 devices	168.07	166.099	166.92
2 Protocol actor running with 40 devices	173.71	166.88	170.64
3 Protocol actor running with 10 devices	196.28	204.23	215.2
3 Protocol actor running with 40 devices	197.34	193.05	196.63

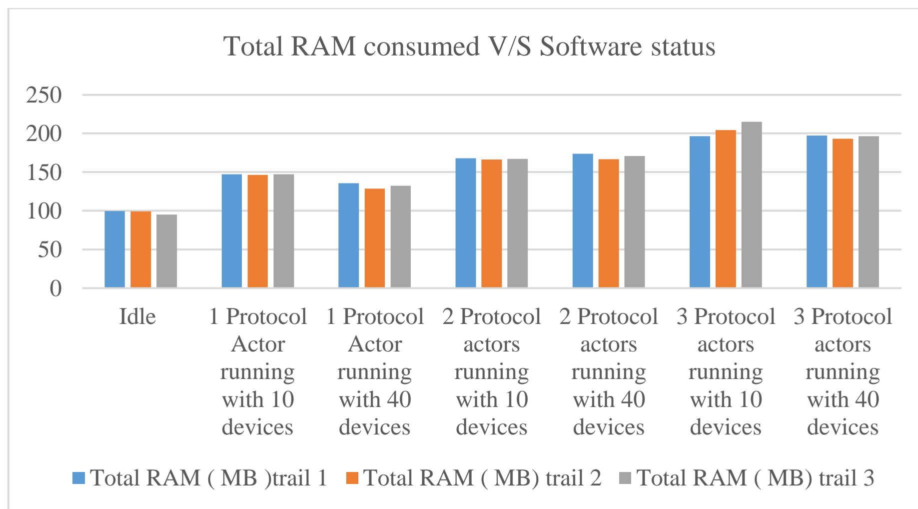

Figure 15: RAM usage vs. number of processes running

5. CONCLUSION AND FUTURE WORK

A cloud-based BEM system is the only solution to address scalability and ease of use issues faced by conventional BEM software. A cloud-based BEM software in comparison with conventional BEM software has unlimited hardware resources and supports desired maintenance-free installation. Hence, migration of BEM software functionalities to the cloud would solve all drawbacks of conventional BEM systems.

Although migration of BEM software to the cloud offers accessibility to recently developed smart devices, it has innate issues addressing legacy devices. A cloud-based BEM software can only connect to devices in its direct reach, i.e., Internet, but cannot reach to devices residing on a local area network with private IP addresses. A naive solution to make these devices accessible to the cloud-based BEM software would be to port forward these devices in case a router supports port forwarding. Such an approach is manual and vulnerable to a number of issues concerning security and privacy. Hence, an IoT gateway is proposed as a solution to address this issue.

An IoT gateway resides in a local area network and thereby can connect to devices within a network. This device can discover, monitor and control any device with legacy protocols by acting as a master device in the network. In order to be compatible with the cloud-based BEM software framework, this device by itself does not initiate any request. Once set up, it maintains a persistent connection with the cloud-based BEM software and addresses any requests from the cloud-based BEM software. As it receives a request, it parses the request and makes an appropriate request to the devices in the network. Hence, this device acts as an edge device and empowers the cloud-based BEM software to access any supported device in a local area network.

The developed IoT gateway is based on the NAT traversal mechanism which is independent of any router specifications and hence with no firewall restrictions imposed on traffic; it can be assessed by a cloud-based BEM software with ease. It is also developed with the most up-to-date technologies and models in order to achieve a high rate of concurrency at a scalable note. With the current setup, in total, a maximum of six processes, i.e., the Actor System, the Client actor, the Handler actor, as well as protocol actors for Modbus, BACnet and RESTful, are executed at any given time to process cloud-based BEM software requests. Support for any device adds a bit to memory per process which is very efficient than forking a new process for every new device. Additionally, since each actor is scheduled by the operating system, they can run on any number of hardware cores, thereby, achieve parallelism. Hence, lightweight, highly concurrent, non-blocking, scalable framework is designed with a minimum amount of resource consumption.

Performance tests have concluded that the amount of RAM usage does not vary even after each protocol supports 50 devices. RAM or resource usage of developed IoT gateway increases only with the addition of new protocols. Currently, the software can comfortably run on a 500 MB constrained hardware resource if hosted on a lightweight operating system. Additionally, the proposed architecture has a great room to build upon to commercially deploy.

Some of the future work which can augment the developed IoT gateway functionalities are:

1) Scalable to support a distributed IoT gateway architecture

An actor system can be interfaced with any other actor system running in different host within a local area network or the Internet. Hence, the developed IoT gateway can feature itself to be a node and conform to a distributed monitoring architecture with a central monitoring system. Hence, IoT gateway software is not constrained to work only with the cloud-based BEM software but can also work in conjunction with other resources.

2) Provision to integrate with HTTP server

In order to commercially deploy the developed software, it requires having a local HTTP server running where the user can configure the IoT gateway. Since the designed system is within an actor system, an HTTP server can be interfaced with an actor system. This HTTP server renders HTML pages where a user can configure the IoT gateway.

3) Extensible to integrate with Django framework and support dew computing

Similar to integrating the Actor System with the HTTP server, IoT gateway software can be integrated with Django framework to develop a comprehensive BEM setup. This can perform all functionalities supported by the cloud-based BEM software and hence run as a standalone BEM software.

4) Remote resource management

In the current architecture, the Handler takes care of spawning an actor for each protocol. As the number of protocols or devices increase to a higher number, this architecture can be extended by providing spawning actors only from the cloud-based BEM software. Such an approach would provide a cloud-based BEM software additional privileges to monitor and efficiently manage processes running on the IoT gateway. With this flexibility, a cloud-based BEM software can ensure reliable functionality of the IoT gateway and restart the system in case of any shutdown.

5) Extensible to apply edge analytics

Since, the IoT gateway software is developed to have only a single point entry from which data passes inside and outside, a data monitoring system can be interfaced to perform any kind of edge analytics or to perform cyber intrusion checks.

6) Extensible to support remote update

Since there exists a persistent connection between the Client and the cloud-based BEM software, an IoT gateway can receive any new configuration files to add support to new devices. Additionally, the entire actor class can be sent in a zip format which is loadable by the actor system.

7) Embedded private key deployment

Since the entire authentication of the IoT gateway is dependent on the private key which is deployed on unsecured environments, private keys must be secured from any physical tampering. For this, the IoT gateway device can be embedded with Trusted Platform Module (TPM) to secure the private keys of all Digital Certificates.

8) Add more support to non-cloud protocols and devices

Since the developed platform is scalable to add support to more devices supporting BACnet, Modbus, and HTTP RESTful protocols. Additionally, support for more non-cloud protocols can be added. With a great range of support, IoT gateway software gets more versatile.

REFERENCES

[1] “Cost of electricity by source.” *Annual Energy Outlook 2017,* 7 Nov. 2012. Web. 25 Nov. 2012. < https://www.eia.gov/outlooks/aeo/ >

[2] Tabors, R.; Parker, G.; Caramanis, M. Development of the Smart Grid: Missing Elements in the Policy Process. In Proceedings of the 2010 43rd Hawaii International Conference on System Sciences (HICSS), Koloa, Kauai, HI, USA, 5–8 January, 2010

[3] https://en.wikipedia.org/wiki/Industrial_control_system

[4] "The Open Market Internet Index". Treese.org. 1995-11-11. Retrieved 2013-06-15.

[5] http://www.gartner.com/newsroom/id/2636073

[6] Shawish, Ahmed, and Maria Salama. "Cloud computing: paradigms and technologies." Inter-cooperative collective intelligence: Techniques and applications. Springer Berlin Heidelberg, 2014. 39-67.

[7] https://dzone.com/articles/IoT-software-platform-comparison

[8] https://en.wikipedia.org/wiki/Network_address_translation

[9] https://en.wikipedia.org/wiki/NAT_traversal

[10] https://developers.google.com/talk/libjingle/important_concepts

[11] http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

[12] Bushby, S.T., 1997. "BACnet: a standard communication infrastructure for intelligent buildings" Automation in Construction 6 (1997). Elsevier, pp. 529-540.

[13] Swan, W., July 1996. "The Language of BACnet" Engineered Systems. Vol. 13, No. 7, pp. 24-32.

[14] Fisher, D.M., July 1996. "BACnet & LonWorks: A White Paper" Private Correspondence.

[15] Swan, W., January 1997. "Internetworking with BACnet" Engineered Systems. Vol. 14, No. 1, pp. 90-99.

[16] Fellows, R.A., March 2000. "Connecting BACnet to the Internet" HPAC - Heating/Piping/Air Conditioning. Vol. 72, No. 3, pp. 65-71

[17] http://events.linuxfoundation.org/sites/events/files/slides/Build%20a%20Micro%20HTTP%20Server% 20for%20Embedded%20System.pdf

[18] Asim, Makkad. "A Survey on Application Layer Protocols for Internet of Things (IoT)." International Journal 8.3 (2017).

[19] Sinha, Sudhi R., and Youngchoon Park. "Creating Smart Gateway." Building an Effective IoT Ecosystem for Your Business. Springer, Cham, 2017. 37-47.

[20] Jung, Markus, et al. "A transparent ipv6 multi-protocol IoT gateway to integrate building automation systems in the Internet of things." Green Computing and Communications (GreenCom), 2012 IEEE International Conference on. IEEE, 2012.

[21] Emara, K. A., Abdeen, M., & Hashem, M. A IoT gateway-based framework for transparent interconnection between WSN and IP network. In EUROCON '09, pages 1775-1780.

[22] Qian, Z., Ruicong, W., Qi, C., Yan, L., & Weijun, Q. IoT Gateway: Bridging Wireless Sensor Networks into Internet of Things. In 2010 IEEE/IFIP 8th International Conference on the Embedded and Ubiquitous Computing (EUC'2010), pages 347-352

[23] Park, KyungGyu, et al. "Building energy management system based on smart grid." Telecommunications Energy Conference (INTELEC), 2011 IEEE 33rd International. IEEE, 2011.

[24] Vresk, Tomislav, and Igor Čavrak. "Architecture of an interoperable IoT platform based on microservices." Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on. IEEE, 2016.

[25] Desai, Pratikkumar, Amit Sheth, and Pramod Anantharam. "Semantic IoT gateway as a service architecture for IoT interoperability." Mobile Services (MS), 2015 IEEE International Conference on. IEEE, 2015.

[26] Guoqiang, Shang, et al. "Design and implementation of a smart IoT gateway." Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing. IEEE, 2013.

[27] Datta, Soumya Kanti, Christian Bonnet, and Navid Nikaein. "An IoT gateway centric architecture to provide novel M2M services." Internet of Things (WF-IoT), 2014 IEEE World Forum on. IEEE, 2014.

[28] Zhu, Qian, et al. "IoT gateway: Bridgingwireless sensor networks into Internet of things." Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on. IEEE, 2010

[29] Ungurean, Ioan, Nicoleta-Cristina Gaitan, and Vasile Gheorghita Gaitan. "An IoT architecture for things from industrial environment." Communications (COMM), 2014 10th International Conference on. IEEE, 2014.

[30] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the Internet of things. In workshop on Mobile cloud computing. ACM, 2012.

[31] Aazam, Mohammad, and Eui-Nam Huh. "Fog computing and smart IoT gateway based communication for cloud of things." Future Internet of Things and Cloud (FiCloud), 2014 International Conference on. IEEE, 2014.

[32] Datta, Soumya Kanti, Christian Bonnet, and Jerome Haerri. "Fog Computing architecture to enable consumer centric Internet of Things services." Consumer Electronics (ISCE), 2015 IEEE International Symposium on. IEEE, 2015.

[33] "What Comes After the Cloud? How About the Fog?". IEEE Spectrum: Technology, Engineering, and Science News. Retrieved 2017-04-07.

[34] "Is There a Buzz Over Fog Computing?". Channelnomics. Retrieved 2017-04-07.

[35] "New Solutions on the Horizon— "Fog" or "Edge" Computing?". The National Law Review. Retrieved 2017-04-07.

[36] Arkian, Hamid Reza; Diyanat, Abolfazl; Pourkhalili, Atefe (2017-03-15). "MIST: Fog-based data analytics scheme with cost-efficient resource provisioning for IoT crowd sensing applications". Journal of Network and Computer Applications. 82: 152–165. doi:10.1016/j.jnca.2017.01.012

[37] Cisco RFP-2013-078. Fog Computing, Ecosystem, Architecture and Applications:

[38] https://qrl.dell.com/Files/en-us/Html/Z5000%20eval/Spec%20Sheet.html

[39] https://www.intel.com/content/dam/www/public/us/.../IoT gateway-solutions-IoT-brief.pdf

[40] https://www.cisco.com/c/dam/en/us/.../internet-of-things/brochure-c02-734481.pdf

[41] https://en.wikipedia.org/wiki/Actor_model

[42] http://thespianpy.com/doc/in_depth.pdf

[43] http://thespianpy.com/

[44] https://en.wikipedia.org/wiki/Transport_layer

[45] https://security.stackexchange.com/questions/20803/how-does-ssl-tls-work

[46] https://github.com/JoelBender/bacpypes

[47] https://github.com/riptideio/pymodbus