MALLA REDDY INSTITUTE OF TECHNOLOGY AND SCIENCE

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

LECTURE MATERIAL

ON

INTERNET OF THINGS

R16 B.Tech ECE IV Year I Sem.

PREPARED BY

Mr. M. ROHITH ANANTH RATNAM

ASSISTANT PROFESSOR

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

MALLA REDDY INSTITUTE OF TECHNOLOGY AND SCIENCE

Permanently affiliated to JNTUH and approved by AICTE, New Delhi

NBA Accredited for B. Tech, ISO 9001:2015 certified, approved by U. K Accreditation centre, Granted status of 2(f) and 12(b) under UGC act 1956, Govt. of India.

MAISAMMAGUDA, DHULAPALLY, SECUNDERABAD-500100

INTERNET OF THINGS
(PROFESSIONAL ELECTIVE – III)

B.Tech. IV Year I Sem. **Course Code: EC732PE**

Course Objectives:

- To introduce the terminology, technology and its applications
- To introduce the concept of M2M (machine to machine) with necessary protocols
- To introduce the Python Scripting Language which is used in many IoT devices
- To introduce the Raspberry PI platform, that is widely used in IoT applications
- To introduce the implementation of web-based services on IoT devices.

UNIT - I

Introduction to Internet of Things -Definition and Characteristics of IoT, Physical Design of IoT – IoT Protocols, IoT communication models, IoT Communication APIs, IoT enabled Technologies – Wireless Sensor Networks, Cloud Computing, Big data analytics, Communication protocols, Embedded Systems, IoT Levels and Templates, Domain Specific IoTs – Home, City, Environment, Energy, Retail, Logistics, Agriculture, Industry, health and Lifestyle.

UNIT - II

IoT and M2M - Software defined networks, network function virtualization, difference between SDN and NFV for IoT. Basics of IoT System Management with NETCOZF, YANG - NETCONF, YANG, SNMP NETOPEER

UNIT - III

Introduction to Python - Language features of Python, Data types, data structures, Control of flow, functions, modules, packaging, file handling, data/time operations, classes, Exception handling. Python packages - JSON, XML, HTTP Lib, URL Lib, SMTP Lib.

UNIT - IV

IoT Physical Devices and Endpoints - Introduction to Raspberry PI - Interfaces (serial, SPI, I2C). Programming – Python program with Raspberry PI with focus of interfacing external gadgets, controlling output, reading input from pins.

UNIT - V

IoT Physical Servers and Cloud Offerings - Introduction to Cloud Storage models and communication APIs. Web server – Web server for IoT, Cloud for IoT, Python web application framework. Designing a REST ful web API.

TEXT BOOK:

1. Internet of Things - A Hands-on Approach, Arshdeep Bahga and Vijay Madisetti, Universities Press, 2015, ISBN: 9788173719547
2. Getting Started with Raspberry Pi, Matt Richardson & Shawn Wallace, O'Reilly (SPD), 2014, ISBN: 9789350239759.

UNIT – I
INTRODUCTION TO INTERNET OF THINGS

Introduction:

The concept of a network of smart devices was discussed as early as 1982, with a modified Coke machine at Carnegie Mellon University becoming the first internet-connected appliance, able to report its inventory and whether newly loaded drinks were cold. Kevin Ashton (born 1968) is a British technology pioneer who is known for inventing the term "the Internet of Things" to describe a system where the Internet is connected to the physical world via ubiquitous sensors. IoT is able to interact without human intervention. Some preliminary IoT applications have been already developed in healthcare, transportation, and automotive industries. IoT technologies are at their infant stages; however, many new developments have occurred in the integration of objects with sensors in the Internet. The development of IoT involves many issues such as infrastructure, communications, interfaces, protocols, and standards. The objective of this paper is to give general concept of IoT, the architecture and layers in IoT, some basic terms associated with it and the services provided. The below fig 1.1 give an example things connected to internet.

The IOT concept was coined by a member of the Radio Frequency Identification (RFID) development community in 1999, and it has recently become more relevant to the practical world largely because of the growth of mobile devices, embedded and ubiquitous communication, cloud computing and data analytics

Fig 1.1: Things connected to Internet.

Definition of IoT:

The Internet of Things (IoT) is the network of physical objects—devices, instruments, vehicles, buildings and other items embedded with electronics, circuits , software, sensors and network connectivity that enables these objects to collect and exchange data. The Internet of Things allows objects to be sensed and controlled remotely across existing network infrastructure, creating opportunities for more direct integration of the physical world into computer-based systems, and resulting in improved efficiency and accuracy.

IoT refers to the interconnection via the internet of computing devices embedded in everyday objects, enabled them to send and receive the data.

A **dynamic global network** infrastructure with **self- configuring capabilities** based on standard and **interoperable communication protocols,** where physical and virtual "things" have **identities,** physical attributes, and use intelligent interfaces, and are seamlessly **integrated into information network** that communicate data with users and environments.

Characteristics of IoT :

1. **Dynamic & Self-Adapting:** IoT device and system may have the capability to dynamically adapt with the changing contexts and take actions based on their operating conditions, user's context, or sensed environment. For example, consider a surveillance adpt their modes based on the weather it is day or night, cameras could switch from lower resolution to higher resolution modes when any motion is detected and alert nearby cameras to do the same.
2. **Self-Configuring:** IoT devices may have self-configuring capability, allowing a large number of devices to work together to provide certain functionality (such as weather monitoring). These devices have the ability configure themselves, setup the networking and fetch latest software upgrades with minimal manual or user intervention.
3. **Interoperable Communication Protocols:** IoT devices may support a number of interoperable communication protocols and can communicate with other devices and also with the infrastructure.
4. **Unique Identity:** Each IoT device has a unique identity and a unique identifier (such as an IP address). IoT systems may have intelligent interface which adapt based on the context, allow communicating with user and the environmental contexts, IoT device interfaces allow users to query the devices, monitor their status and control them remotely.
5. **Integrated into Information Network:** IoT devices are usually integrated into the information network that allows them to communicate and exchange data with other devices and systems, IoT dvices can be dynamically discovered in the network, by other devices and/or the network, and have the capability to describe themselves to other devices or user applications.

Physical Design of IoT

Things in IoT

1. Refers to IoT devices which have unique identities that can perform sensing, actuating and monitoring capabilities.
2. IoT devices can exchange data with other connected devices or collect data from other devices and process the data either locally or send the data to centralized servers or cloud – based application back-ends for processing the data.

Generic Block Diagram of an IoT Device:

An IoT device may consist of several interfaces for connections to other devices, both wired and wireless. The below Fig 1.2 shows the block diagram of an IoT Device.

- I/O interfaces for sensors
- Interfaces for internet connectivity
- Memory and storage interfaces
- Audio/video interfaces

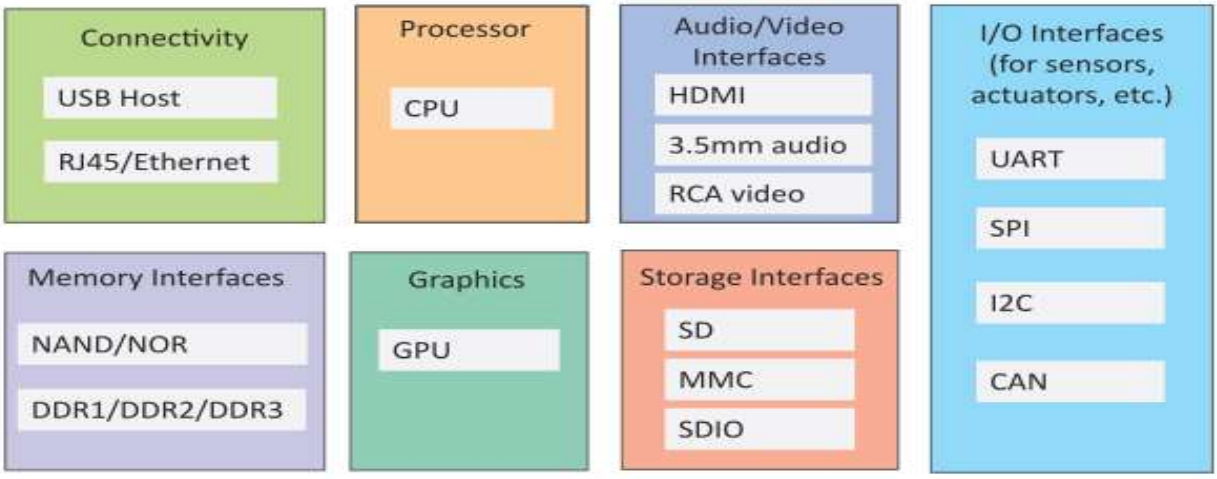


Fig 1.2: Block Diagram of an IoT Device

IoT Protocols:

The IoT devices are typically connected to the Internet via an IP (Internet Protocol) network. However, devices such as Bluetooth and RFID allow IoT devices to connect locally. In these cases, there's a difference in power, range, and memory used. Connection through IP networks

are comparatively complex, requires increased memory and power from the IoT devices while the range is not a problem. On the other hand, non-IP networks demand comparatively less power and memory but have a range limitation.

As far as the IoT communication protocols or technologies are concerned, a mix of both IP and non-IP networks can be considered depending on usage.

Fig 1.3 shows Four Layer of IoT Protocol

1. Link Layer
2. Network Layer
3. Transport Layer
4. Application Layer

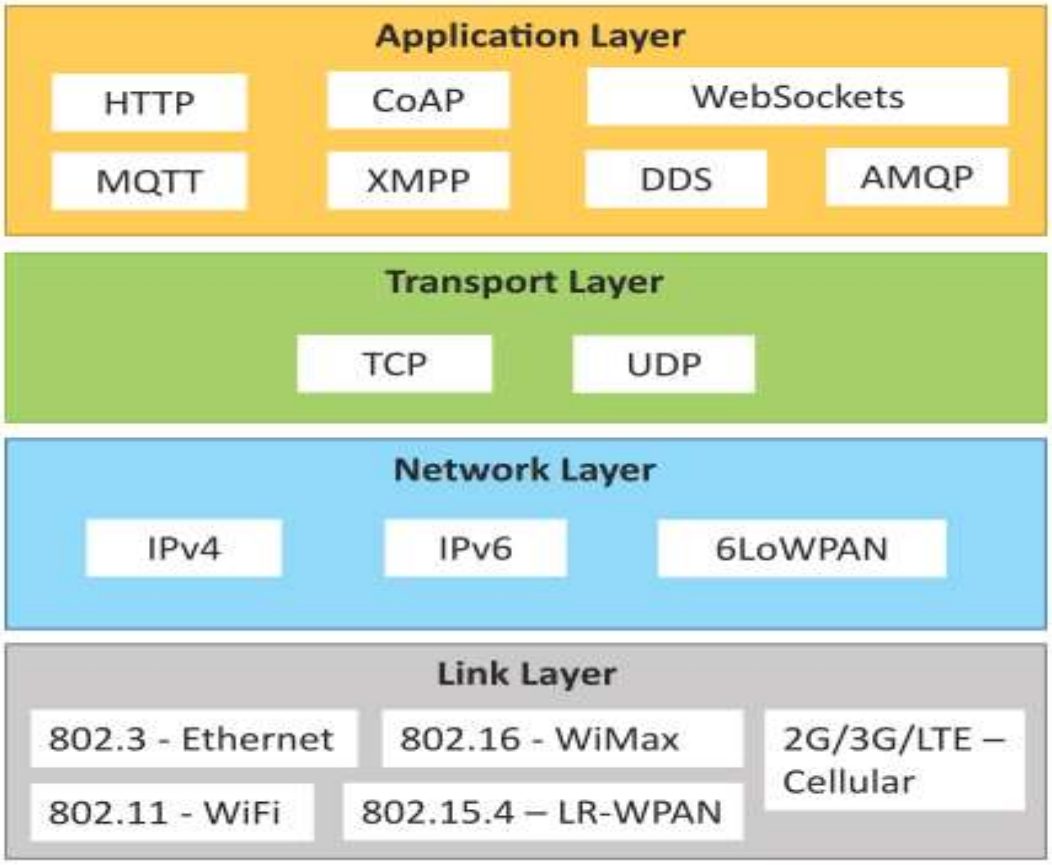


Fig 1.3 Four Layer IoT Protocol

1. Link Layer:

In computer networking, the link layer is the lowest layer in the Internet protocol suite, the networking architecture of the Internet. The link layer is the group of methods and communications protocols confined to the link that a host is physically connected to. The link is the physical and logical network component used to interconnect hosts or nodes in the network

and a link protocol is a suite of methods and standards that operate only between adjacent network nodes of a network segment.

Table 1.1: different methods of link layer with standards.

Ethernet Standard		
Sr.No	**Standard**	**Shared medium**
1	802.3	Coaxial cable
2	802.3.i	Copper Twisted pair
3	802.3.j	Fiber Optic
4	802.3.ae	Fiber…..10Gbits/s
WiFi Standard		
S.No	Standard	Operates in
1	802.11a	5 GHz band
2	802.11b& 802.11g	2.4GHz band
3	802.11.n	2.4/5 GHz bands
4	802.11.ac	5GHz band
5	802.11.ad	60Hz band
WiMax Standard		
S.No	Standard	Data Rate
1	802.16m	100Mb/s for mobile stations, 1Gb/s for fixed stations
Mobile Communication Standard		
Sr.No	Standard	Operates in
1	2G	GSM-CDMA
2	3G	UMTS and CDMA 2000
3	4G	LTE

The above Table 1.1 shows the different method of link layer with different standards. For Ethernet method Data Rates are provided from 10Gbit/s to 40Gb/s and higher. Collection of Wireless LAN Data Rates from 1Mb/s to 6.75 Gb/s. Collection of Wireless Broadband standards Data Rates from 1.5Mb/s to 1 Gb/s. LR-WPAN: Collection of standards for low-rate wireless personal area networks, Basis for high level communication protocols such as Zigbee, Data Rates from 40Kb/s to 250Kb/s. 2G/3G/4G –Mobile Communication: Data Rates from 9.6Kb/s (for 2G) to up to 100Mb/s (for 4G).

2. Network/Internet Layer:

The internet layer is a group of internetworking methods, protocols, and specifications in the Internet protocol suite that are used to transport network packets from the originating host across network boundaries; if necessary, to the destination host specified by an IP address. The internet layer derives its name from its function facilitating internetworking, which is the concept of connecting multiple networks with each other through gateways.

- Responsible for sending of IP datagrams from source to destination network
- Performs the host addressing and packet routing
- Host identification is done using hierarchical IP addressing schemes such as IPV4 or IPV6

IPV4

Used to identify the devices on a network using hierarchical addressing scheme
Uses 32-bit address scheme

IPV6

Uses 128-bit address scheme

6LoWPAN (IPV6 over Low power Wireless Personal Area Network)

Used for devices with limited processing capacity, Operates in 2.4 Ghz , Data Rates of 250Kb/s.

3. Transport Layer:

In computer networking, the transport layer is a conceptual division of methods in the layered architecture of protocols in the network stack in the Internet protocol suite and the OSI model. The protocols of this layer provide host-to-host communication services for applications. It provides services such as connection-oriented communication, reliability, flow control, and multiplexing.

The best-known transport protocol of the Internet protocol suite is the Transmission Control Protocol (TCP). It is used for connection-oriented transmissions, whereas the connectionless User Datagram Protocol (UDP) is used for simpler messaging transmissions.

- Provide end-to-end message transfer capability independent of the underlying network
- It provides functions such as error control, segmentation, flow-control and congestion control

Transmission Control Protocol (TCP):

- Connection Oriented
- Ensures Reliable transmission
- Provides Error Detection Capability to ensure no duplicacy of packets and retransmit lost packets
- Flow Control capability to ensure the sending data rate is not too high for the receiver process
- Congestion control capability helps in avoiding congestion which leads to degradation of n/w performance

User Datagram Protocol (UDP):

- Connectionless
- Does not ensures Reliable transmission
- Does not do connection before transmitting
- Does not provide proper ordering of messages
- Transaction oriented and stateless

4. Application Layer:

An application layer is an abstraction layer that specifies the shared communications protocols and interface methods used by hosts in a communications network. The application layer abstraction is used in both of the standard models of computer networking: the Internet Protocol Suite (TCP/IP) and the OSI model. Although both models use the same term for their respective highest level layer, the detailed definitions and purposes are different.

Hyper Transfer Protocol:

- Forms foundation of World Wide Web(WWW)
- Includes commands such as GET,PUT, POST, HEAD, OPTIONS, TRACE..etc
- Follows a request-response model
- Uses Universal Resource Identifiers(URIs) to identify HTTP resources.

Constrained Application Protocol (CoAP):

- Used for Machine to machine (M2M) applications meant for constrained devices and n/w's
- Web transfer protocol for IoT and uses request-response model

- Uses client –server architecture
- Supports methods such as GET,POST, PUT and DELETE

WebSocket:

- Allows full-duplex communication over single socket, Based on TCP, Client can be a browser, IoT device or mobile application

Message Queue Telemetry Transport (MQTT):

- light-weight messaging protocol, Based on publish-subscribe model, Well suited for constrained environments where devices have limited processing, low memory and n/w bandwith requirement

XMPP:

- Extensible messaging and presence protocol, For Real time communication and streaming XML data between n/w entities, Used for Applications such as Multi-party chat and voice/video calls.

Logical Design of IoT:

- Logical design of an IoT system refers to an abstract representation of the entities and processes without going into the low-level specifics of the implementation.
- An IoT system comprises a number of functional blocks that provide the system the capabilities for identification, sensing, actuation, communication and management.
- Fig 1.4 shows the functional block diagram of IoT. Blew are the individual block explanation.

1. Device : Devices such as sensing, actuation, monitoring and control functions.
2. Communication : IoT Protocols
3. Services like device monitoring, device control services, data publishing services and device discovery
4. Management : Functions to govern the system
5. Security : Functions as authentication, authorization, message and content integrity, and data security Applications

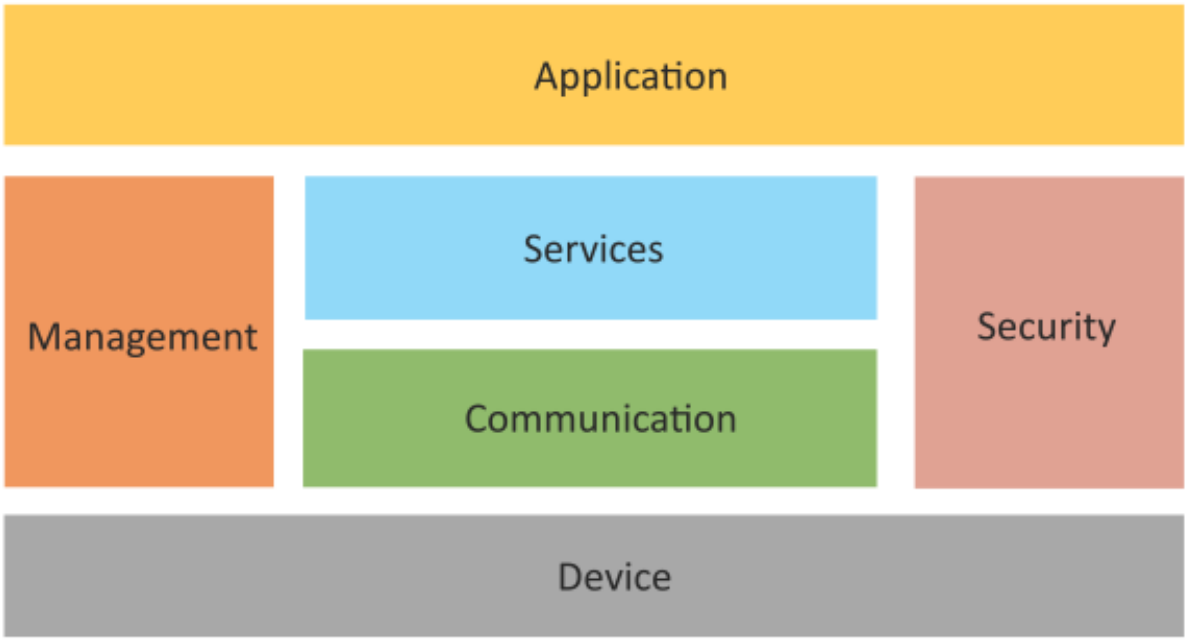


Fig 1.4: Functional Block Diagram of IoT.

IoT Communication Modules:

1. **Request–Response Communication Model:**
   - Request–Response is a communication model in which the client sends requests to the server and the server responds to the requests.
   - When the server receives a request, it decides how to respond, fetches the data, retrieves resource representations, prepares the response and then sends the response to the client.
   - Stateless communication model, Fig 1.5 shows the block diagram of Request-Response Communication Model.

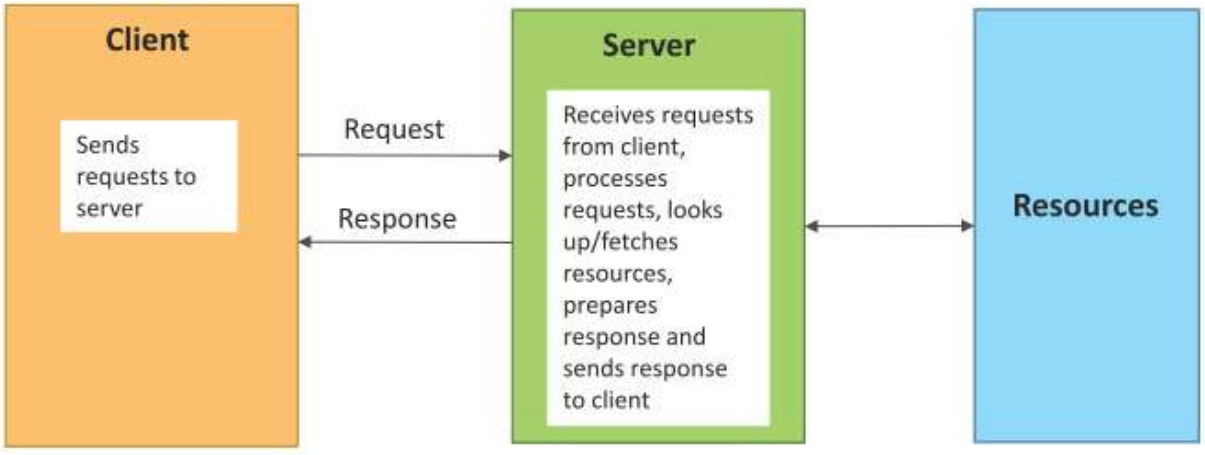


Fig 1.5: Request–Response Communication Model.

2. Publish–Subscribe Communication Model:

- Publish–Subscribe is a communication model that involves publishers, brokers and consumers.
- Publishers are the source of data. Publishers send the data to the topics which are managed by the broker. Publishers are not aware of the consumers.
- Consumers subscribe to the topics which are managed by the broker.
- When the broker receives data for a topic from the publisher, it sends the data to all the subscribed consumers. Fig 1.6 shows the block diagram of Publish-Subscribe Communication Model.

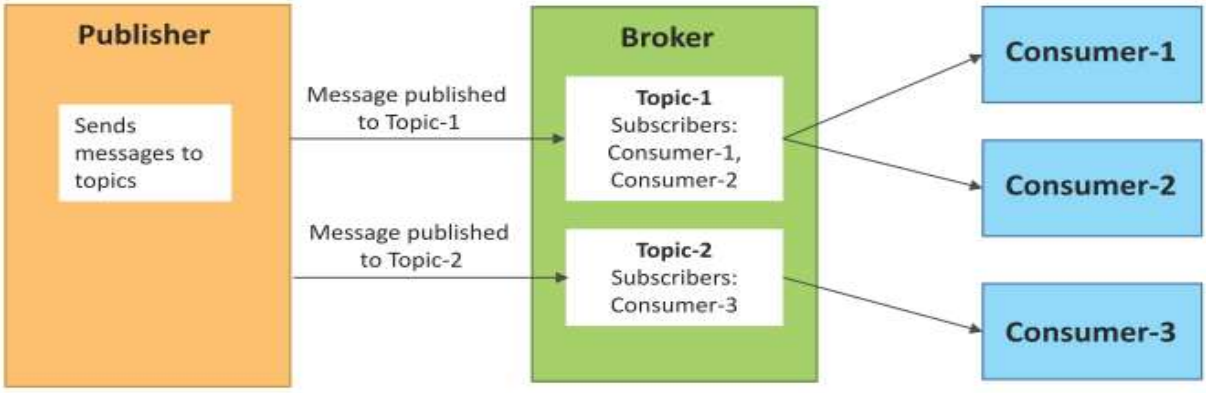


Fig 1.6: Publish–Subscribe Communication Model.

3. Push–Pull Communication Model:

- Push–Pull is a communication model in which the data producers push the data to queues and the consumers pull the data from the queues. Producers do not need to be aware of the consumers.
- Queues help in decoupling the messaging between the producers and consumers.
- Queues also act as a buffer which helps in situations when there is a mismatch between the rate at which the producers push data and the rate at which the consumers pull data. Fig 1.7 shows the block diagram of Push-Pull Communication Model.

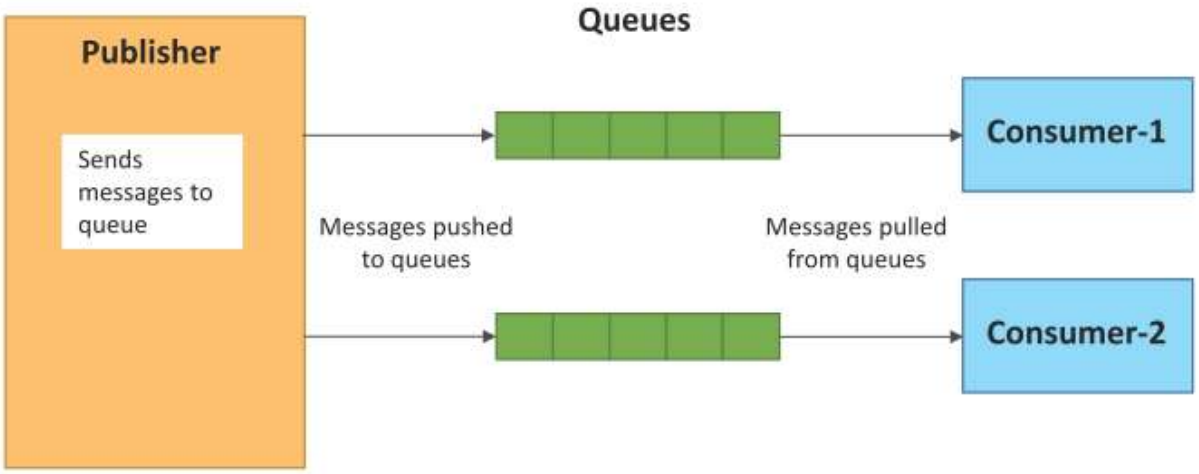


Fig 1.7: Push–Pull Communication Model.

4. Exclusive Pair Communication Model:

- Exclusive Pair is a bidirectional, fully duplex communication model that uses a persistent connection between the client and the server.
- Once the connection is set up it, remains open until the client sends a request to close the connection.
- Client and server can send messages to each other after connection setup. Fig 1.8 shows the block diagram of Exclusive Pair Communication Model

Fig 1.8: Exclusive Pair Communication Model.

IoT Communication APIs:

1. REST-based Communication APIs:

- Representational State Transfer (REST) is a set of architectural principles by which you can design web services and web APIs that focus on a system's resources and how resource states are addressed and transferred.
- REST APIs follow the request–response communication model. Fig 1.9 shows the RESET API Architecture diagram.
- REST architectural constraints apply to the components, connectors and data elements within a distributed hypermedia system.

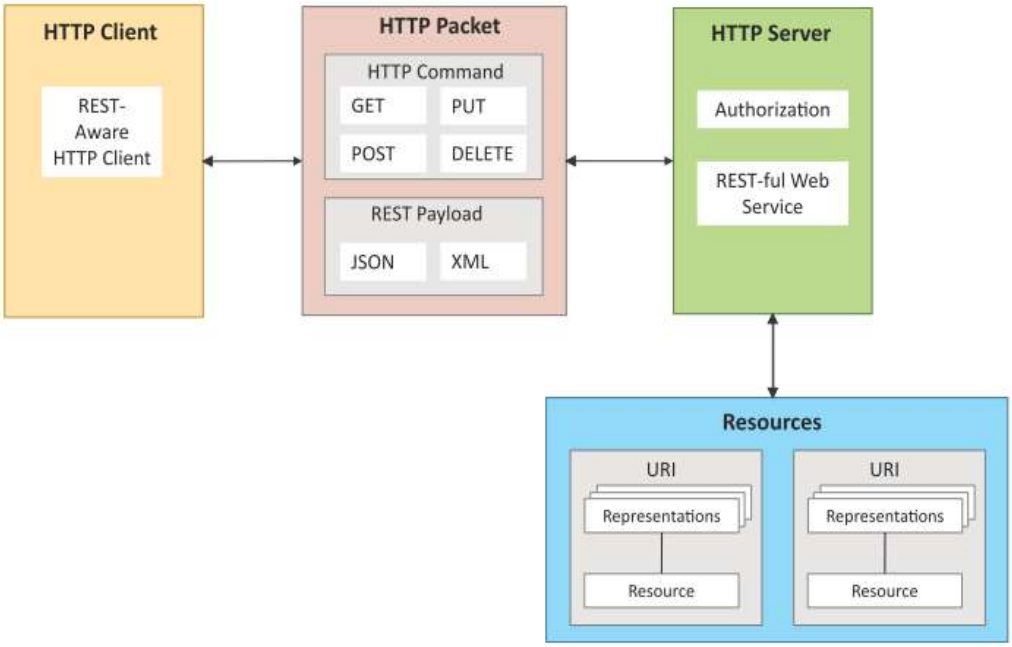


Fig 1.9: REST API Architecture.

Constraints:

- Client – Server
- Stateless
- Cacheable
- Layered System
- Uniform Interface
- Code on demand

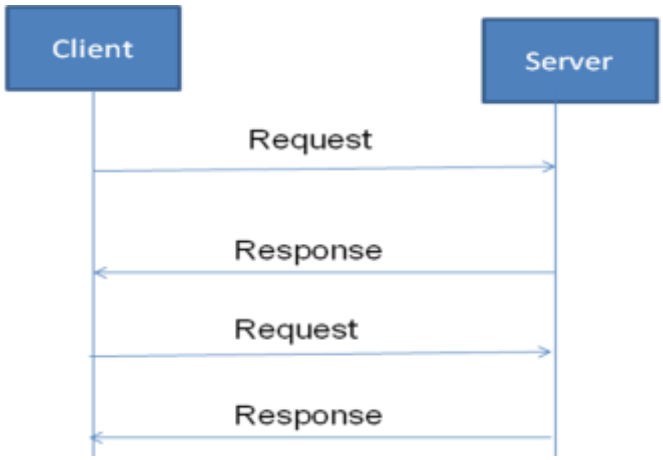


Fig 1.10: RESET API Constrants.

2. Web Socket-based Communication APIs:

- WebSocket APIs allow bi-directional, full duplex communication between clients and servers.
- WebSocket APIs follow the exclusive pair communication model.

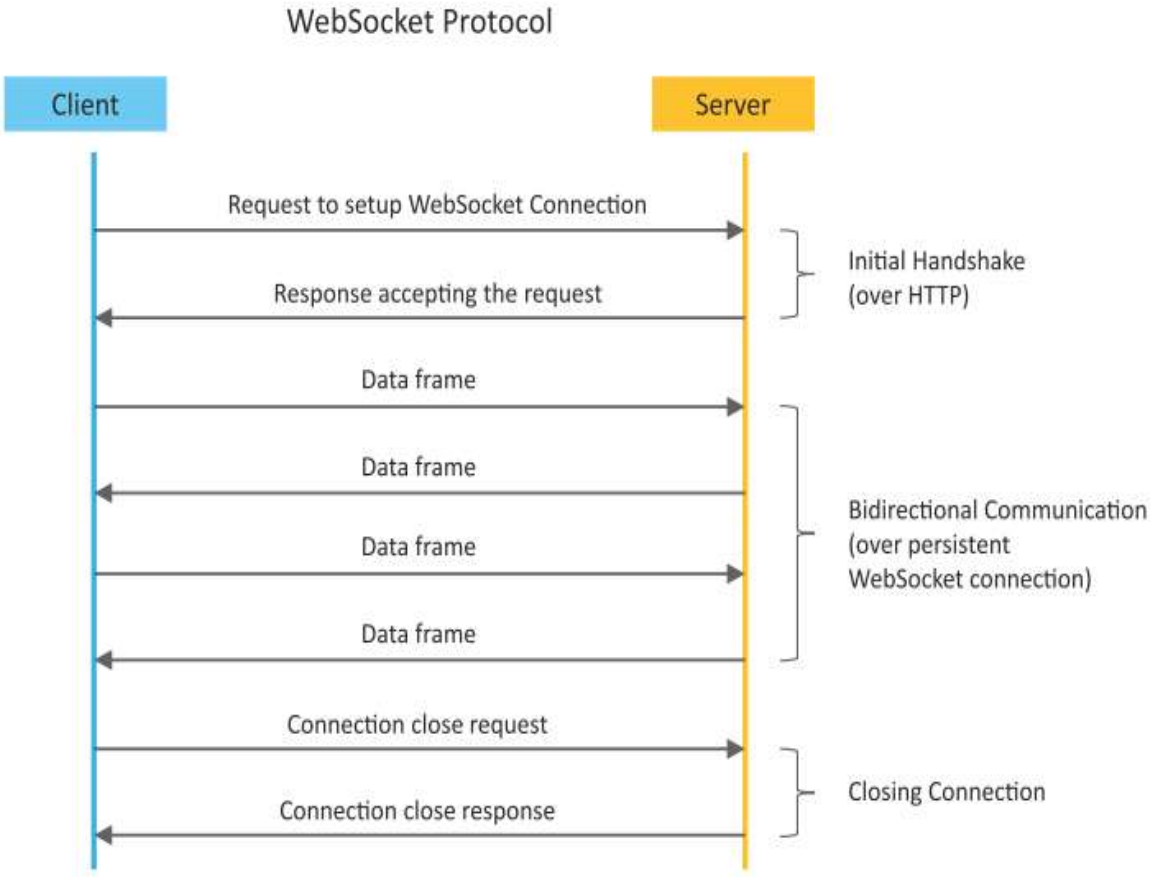


Fig1.11 WebSocket Protocol.

Difference between REST and WebSocket-based Communication APIs:

Comparison Based on	REST	Websocket
State	Stateless	Stateful
Directional	Unidirectional	Bidirectional
Req-Res/Full Duplex	Follow Request Response Model	Exclusive Pair Model
TCP Connections	Each HTTP request involves setting up a new TCP Connection	Involves a single TCP Connection for all requests
Header Overhead	Each request carries HTTP Headers, hence not suitable for real-time	Does not involve overhead of headers.
Scalability	Both horizontal and vertical are easier	Only Vertical is easier

IoT Enabling Technologies:

1. **Wireless Sensor Network(WSN):**
    - Distributed Devices with sensors used to monitor the environmental and physical conditions
    - Consists of several end-nodes acting as routers or coordinators too
    - Coordinators collects data from all nodes / acts as gateway that connects WSN to internet
    - Routers route the data packets from end nodes to coordinators.

    Example:
    - Weather monitoring system
    - Indoor Air quality monitoring system
    - Soil moisture monitoring system
    - Survelliance systems
    - Health monitoring systems

    Protocols:
    - Zigbee

Fig 1.12: wireless Sensors.

2. **Cloud Computing:**
    - Deliver applications and services over internet
    - Provides computing, networking and storage resources on demand
    - Cloud computing performs services such as Iaas, Paas and Saas
    - Iaas : Rent Infrastructure
    - Paas : supply an on-demand environment for developing, testing, delivering and managing software applications.
    - Saas : method for delivering software applications over the Internet, on demand and typically on a subscription basis.

Fig 1.13 Cloud Computing Physical Diagram

3. **Big Data Analytics:**
   - Collection of data whose volume, velocity or variety is too large and difficult to store, manage, process and analyze the data using traditional databases.
   - It involves data cleansing, processing and visualization
   - Lots of data is being collected and warehoused
   - Web data, e-commerce
     - purchases at department/ grocery stores
     - Bank/Credit Card transactions
     - Social Network

Fig 1.14 Big Data Analytics Physical Diagram.

Variety Includes different types of data

- Structured, Unstructured, SemiStructured, All of above

Velocity Refers to speed at which data is processed

- Batch, Real-time, STreams

Volume refers to the amount of data

- Terabyte, Records, Transactions, Files, Tables

4. **Embedded Systems:**

An Embedded System is a computer system that has computer hardware and software embedded toperform specific tasks. In constrast to general purpose computers which can perform various typpes of asks, embedded systems are designed to pperform a specific set of tasks. Key components of an embedded system include, microproceessor or microcontroller, memory(RAM, ROM, Cache) and Input/Output Units.

Communication Protocols:

Communication protocols from the backbone of IoT systems and enabled network connectivity and coupling to applications. Communication protocols allow devices to exchange data over the network. In internet protocol session we discussed about various Link, Network, Transport and Application Layer protocols.

IoT Levels and Deployment Templates:

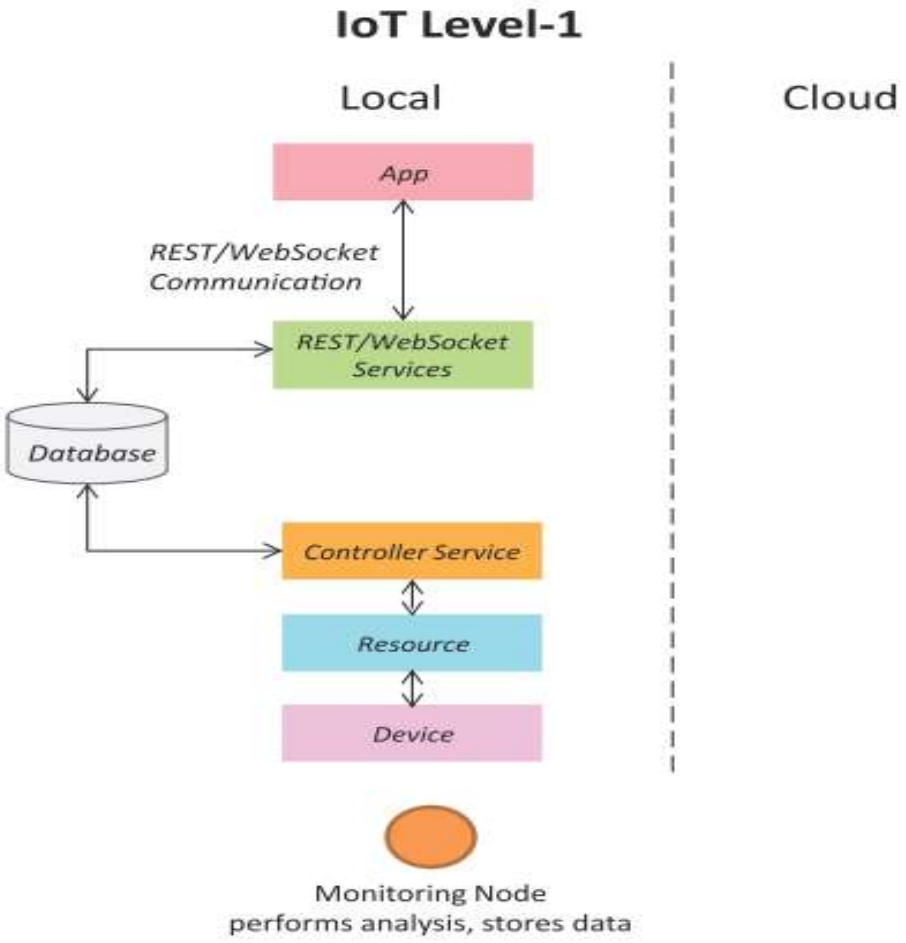
An IoT system comprises the following components:

- **Device**: An IoT device allows identification, remote sensing, actuating and remote monitoring capabilities.
- **Resource**: Resources are software components on the IoT device for accessing, processing and storing sensor information, or for controlling actuators connected to the device. Resources also include the software components that enable network access for the device.
- **Controller Service**: Controller service is a native service that runs on the device and interacts with the web services. Controller service sends data from the device to the web service and receives commands from the application (via web services) for controlling the device.
- **Database**: Database can be either local or in the cloud and stores the data generated by the IoT device.
- **Web Service**: Web services serve as a link between the IoT device, application, database and analysis components. Web service can be implemented using HTTP and REST principles (REST service) or using the WebSocket protocol (WebSocket service).
- **Analysis Component**: This is responsible for analyzing the IoT data and generating results in a form that is easy for the user to understand.
- **Application**: IoT applications provide an interface that the users can use to control and monitor various aspects of the IoT system. Applications also allow users to view the system status and the processed data.

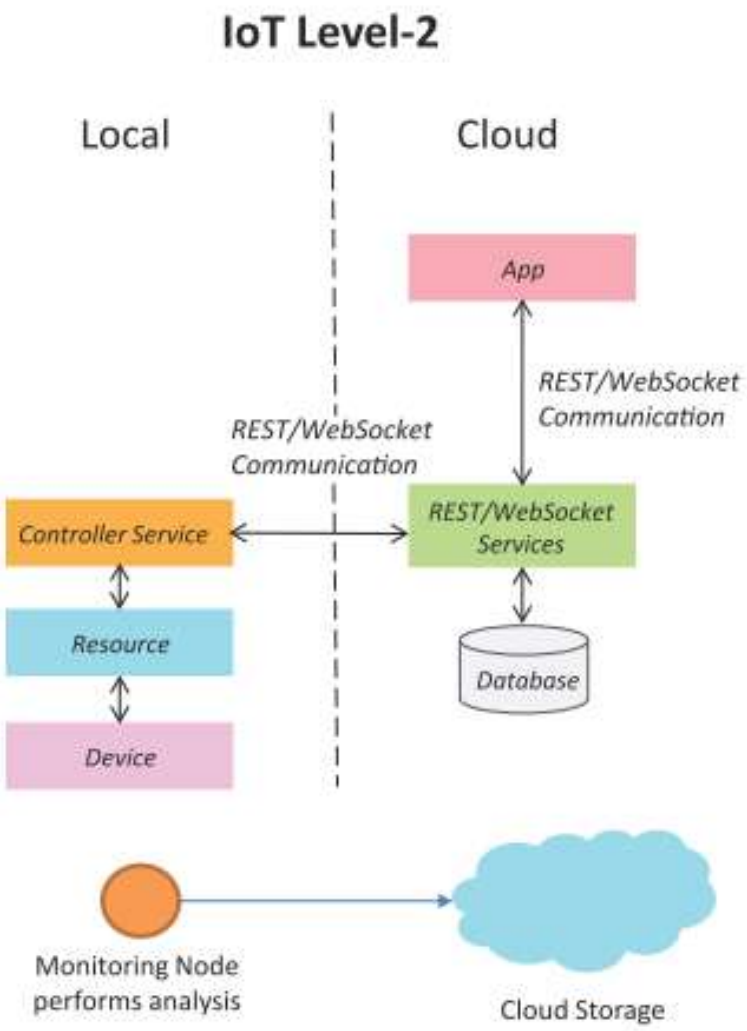
IoT Level-1:

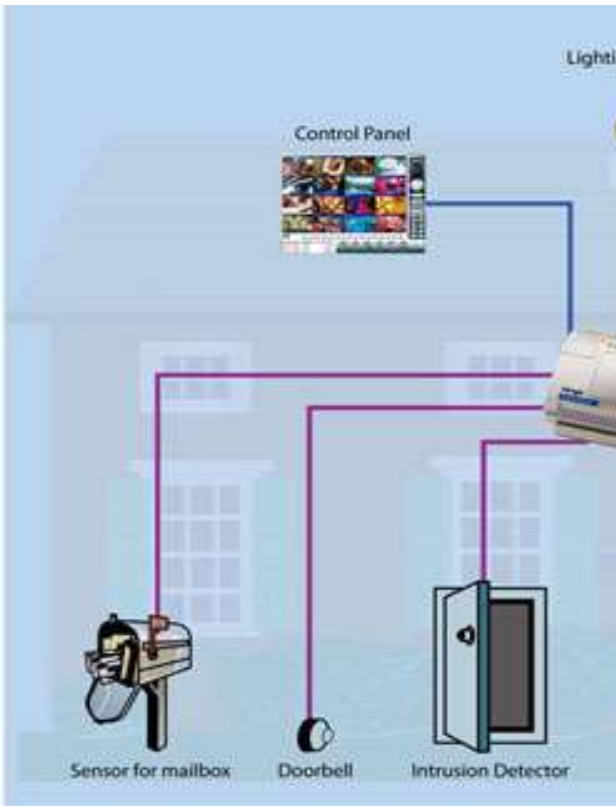
- A level-1 IoT system has a single node/device that performs sensing and/or actuation, stores data, performs analysis and hosts the application.

- Level-1 IoT systems are suitable for modelling low-cost and low-complexity solutions where the data involved is not big and the analysis requirements are not computationally intensive.



Example -Home Automation System:





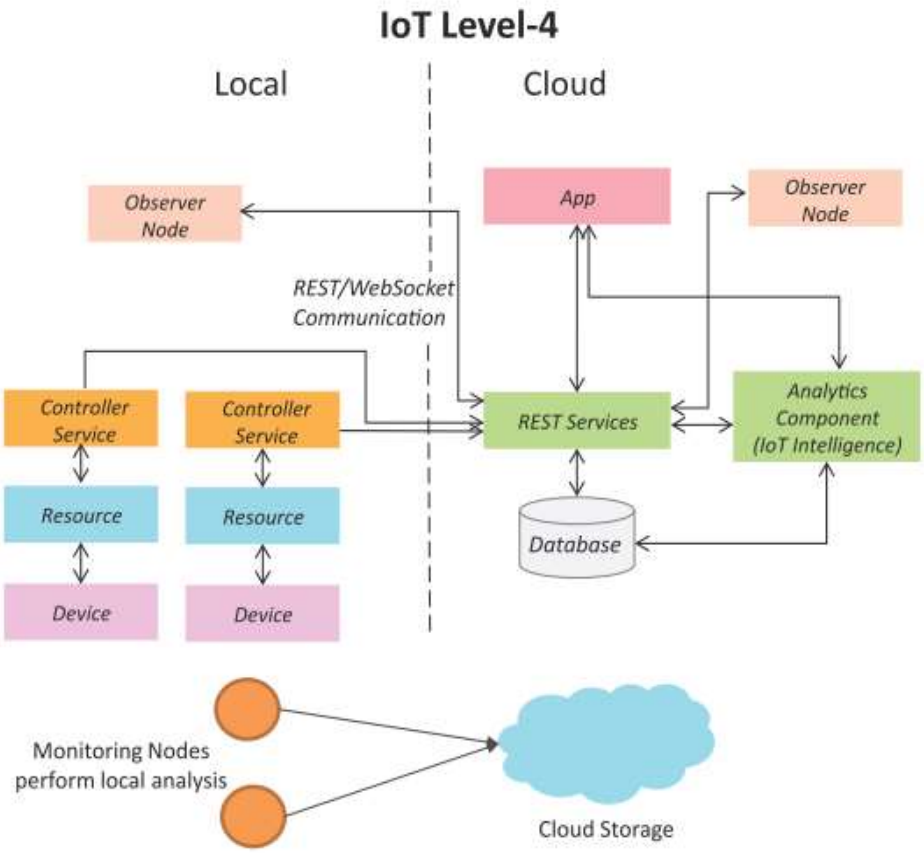
IoT Level-2:

- A level-2 IoT system has a single node that performs sensing and/or actuation and local analysis.
- Data is stored in the cloud and the application is usually cloud-based.
- Level-2 IoT systems are suitable for solutions where the data involved is big; however, the primary analysis requirement is not computationally intensive and can be done locally.

Example-Smart Irrigation:

IoT Level-4:

- A level-4 IoT system has multiple nodes that perform local analysis. Data is stored in the cloud and the application is cloud-based.
- Level-4 contains local and cloud-based observer nodes which can subscribe to and receive information collected in the cloud from IoT devices.
- Level-4 IoT systems are suitable for solutions where multiple nodes are required, the data involved is big and the analysis requirements are computationally intensive.

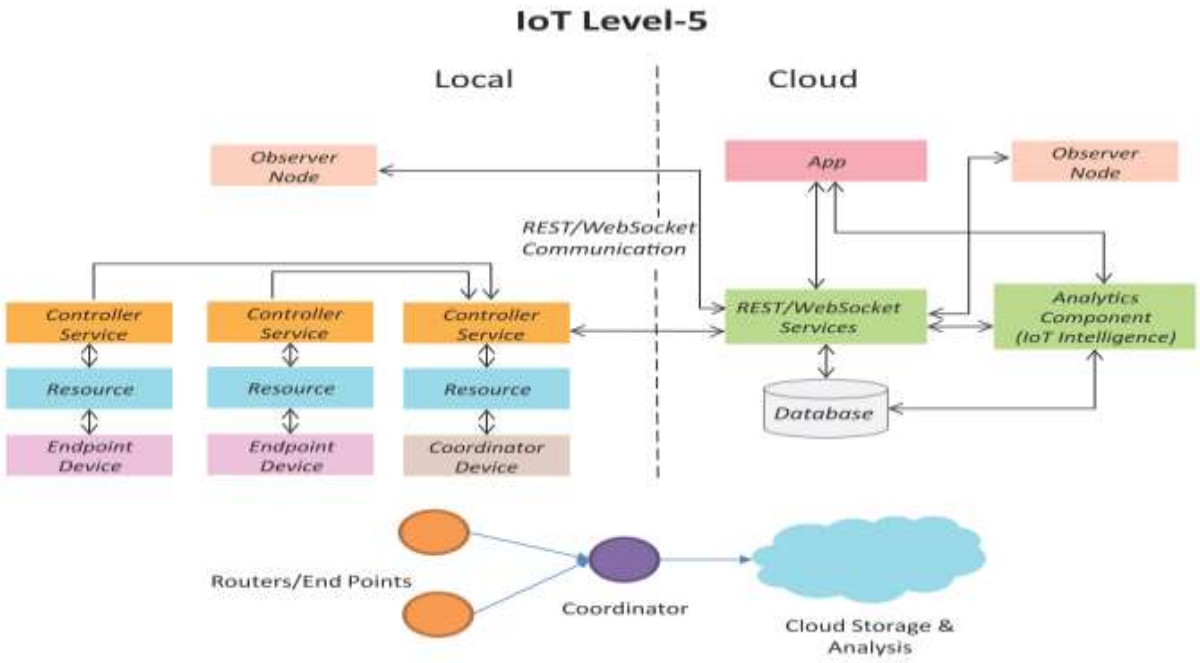


Example-Noise Monitoring:

IoT Level-5:

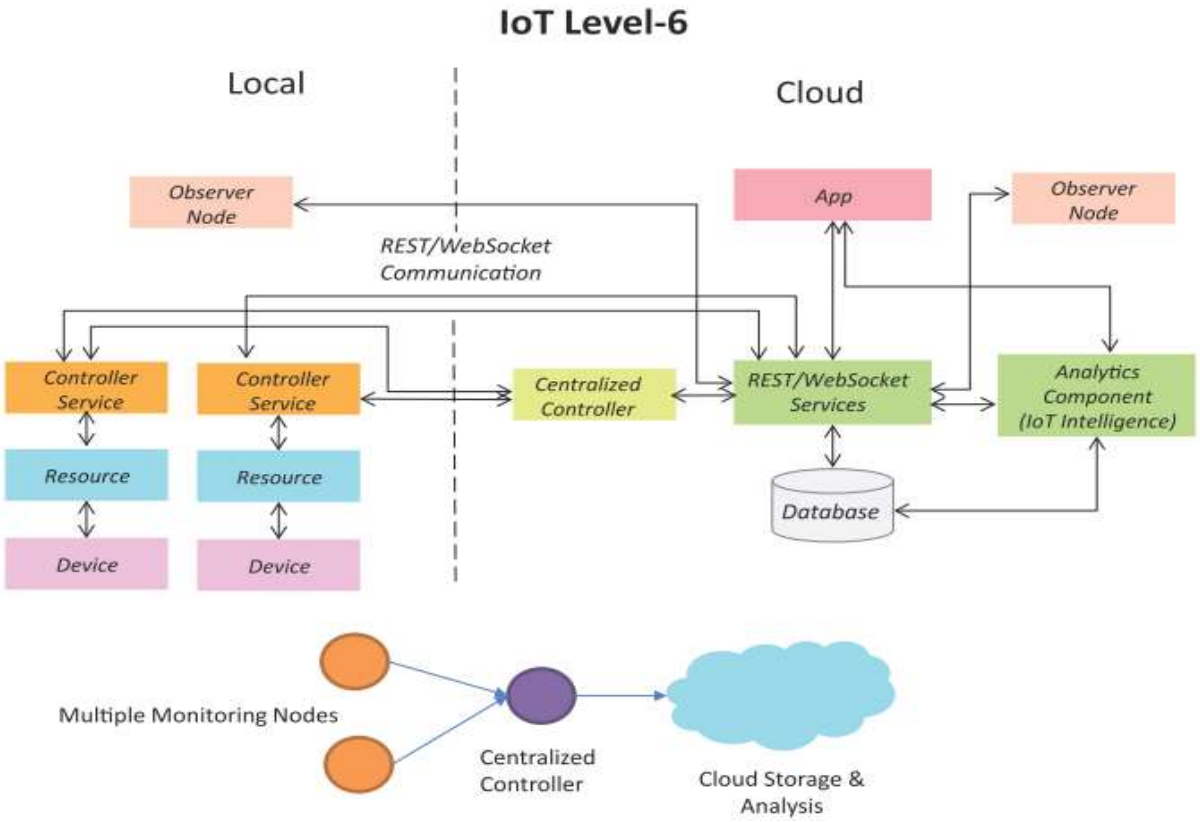
- A level-5 IoT system has multiple end nodes and one coordinator node.

- The end nodes perform sensing and/or actuation.
- The coordinator node collects data from the end nodes and sends it to the cloud.
- Data is stored and analyzed in the cloud and the application is cloud-based.
- Level-5 IoT systems are suitable for solutions based on wireless sensor networks, in which the data involved is big and the analysis requirements are computationally intensive.



IoT Level-6:

- A level-6 IoT system has multiple independent end nodes that perform sensing and/or actuation and send data to the cloud.
- Data is stored in the cloud and the application is cloud-based.
- The analytics component analyzes the data and stores the results in the cloud database.
- The results are visualized with the cloud-based application.
- The centralized controller is aware of the status of all the end nodes and sends control commands to the nodes.



DOMAIN SPECIFIC IoTs

1) Home Automation:

a) Smart Lighting: helps in saving energy by adapting the lighting to the ambient conditions and switching on/off or diming the light when needed.
b) Smart Appliances: make the management easier and also provide status information to the users remotely.
c) Intrusion Detection: use security cameras and sensors (PIR sensors and door sensors) to detect intrusion and raise alerts. Alerts can be in the form of SMS or email sent to the user.
d) Smoke/Gas Detectors: Smoke detectors are installed in homes and buildings to detect smoke that is typically an early sign of fire. Alerts raised by smoke detectors can be in the form of signals to a fire alarm system. Gas detectors can detect the presence of harmful gases such as CO, LPG etc.,

2) Cities:

a) Smart Parking: make the search for parking space easier and convenient for drivers. Smart parking are powered by IoT systems that detect the no. of empty parking slots and send information over internet to smart application back ends.

b) Smart Lighting: for roads, parks and buildings can help in saving energy.

c)Smart Roads: Equipped with sensors can provide information on driving condition, travel time estimating and alert in case of poor driving conditions, traffic condition and accidents.

d) Structural Health Monitoring: uses a network of sensors to monitor the vibration levels in the structures such as bridges and buildings.

e) Surveillance: The video feeds from surveillance cameras can be aggregated in cloud based scalable storage solution.

f) Emergency Response: IoT systems for fire detection, gas and water leakage detection can help in generating alerts and minimizing their effects on the critical infrastructures.

3) Environment:

a) Weather Monitoring: Systems collect data from a no. of sensors attached and send the data to cloud based applications and storage back ends. The data collected in cloud can then be analyzed and visualized by cloud based applications.

b) Air Pollution Monitoring: System can monitor emission of harmful gases (CO2, CO, NO, NO2 etc.,) by factories and automobiles using gaseous and meteorological sensors. The collected data can be analyzed to make informed decisions on pollutions control approaches.

c) Noise Pollution Monitoring: Due to growing urban development, noise levels in cities have increased and even become alarmingly high in some cities. IoT based noise pollution monitoring systems use a no. of noise monitoring systems that are deployed at different places in a city. The data on noise levels from the station is collected on servers or in the cloud. The collected data is then aggregated to generate noise maps.

d) Forest Fire Detection: Forest fire can cause damage to natural resources, property and human life. Early detection of forest fire can help in minimizing damage.

e) River Flood Detection: River floods can cause damage to natural and human resources and human life. Early warnings of floods can be given by monitoring the water level and flow rate. IoT based river flood monitoring system uses a no. of sensor nodes that monitor the water level and flow ratesensors.

4) Energy:

a) Smart Grids: is a data communication network integrated with the electrical grids that collects and analyze data captured in near-real-time about power transmission, distribution and consumption. Smart grid technology provides predictive information and recommendations to utilities, their suppliers, and their customers on how best to manage power. By using IoT based

sensing and measurement technologies, the health of equipment and integrity of the grid can be evaluated.
b) Renewable Energy Systems: IoT based systems integrated with the transformers at the point of interconnection measure the electrical variables and how much power is fed into the grid. For wind energy systems, closed-loop controls can be used to regulate the voltage at point of interconnection which coordinate wind turbine outputs and provides power support.
c) Prognostics: In systems such as power grids, real-time information is collected using specialized electrical sensors called Phasor Measurment Units (PMUs) at the substations. The information received from PMUs must be monitored in real-time for estimating the state of the system and for predicting failures.

5) Retail:
a) Inventory Management: IoT systems enable remote monitoring of inventory using data collected by RFID readers.
b) Smart Payments: Solutions such as contact-less payments powered by technologies such as Near Field Communication (NFC) and Bluetooth.
c) Smart Vending Machines: Sensors in a smart vending machines monitors its operations and send the data to cloud which can be used for predictive maintenance.

6) Logistics:
a) Route generation & scheduling: IoT based system backed by cloud can provide first response to the route generation queries and can be scaled up to serve a large transportation network.
b) Fleet Tracking: Use GPS to track locations of vehicles in real-time.
c) Shipment Monitoring: IoT based shipment monitoring systems use sensors such as temp, humidity, to monitor the conditions and send data to cloud, where it can be analyzed to detect food spoilage.
d) Remote Vehicle Diagnostics: Systems use on-board IoT devices for collecting data on Vehicle operations (speed, RPM etc.,) and status of various vehicle subsystems.

7) Agriculture:
a) Smart Irrigation: to determine moisture amount in soil.
b) Green House Control: to improve productivity.

8) Industry:
a)Machine diagnosis and prognosis
b) Indoor Air Quality Monitoring

9) Health and Life Style:

a) Health & Fitness Monitoring

b) Wearable Electro

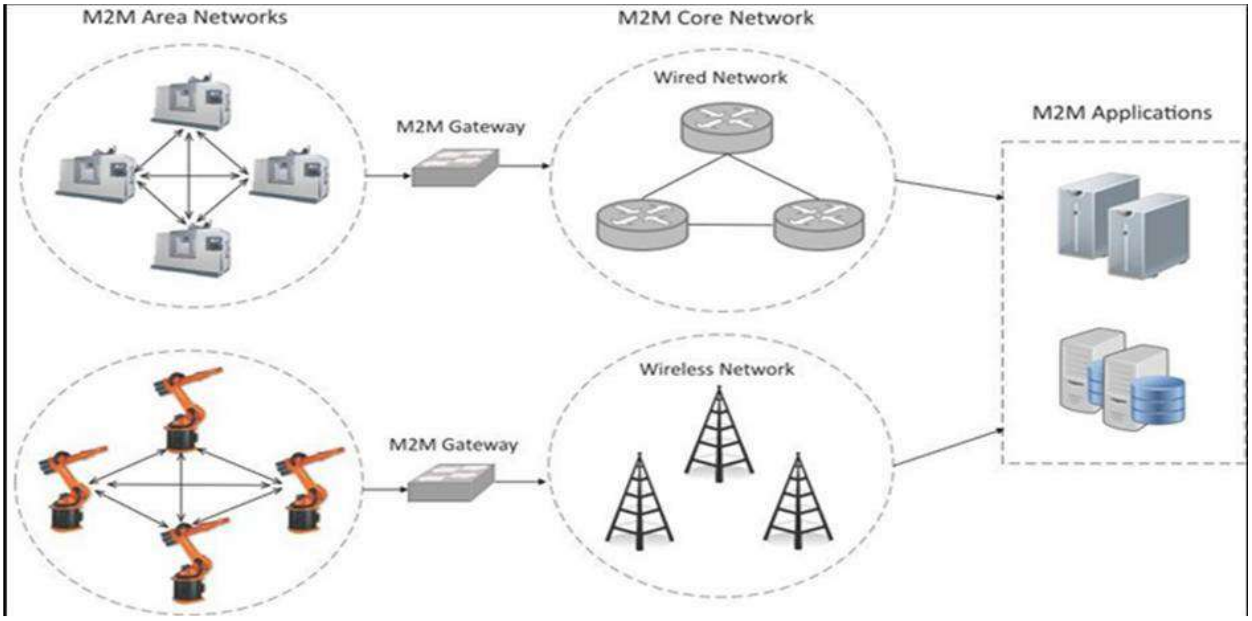
UNIT - II

INTRODUCTION TO M2M:

Machine-to-Machine (M2M) refers to networking of machines (or devices) for the purpose of remote monitoring and control and data exchange.

- Term which is often synonymous with IoT is Machine-to-Machine (M2M).
- IoT and M2M are often used interchangeably.

Fig. Shows the end-to-end architecture of M2M systems comprises of M2M area networks, communication networks and application fomain.



- An M2M area network comprises of machines (or M2M nodes) which have embedded network modules for sensing, actuation and communicating various communication protocols can be used for M2M LAN such as ZigBee, Bluetooth, M-bus, Wireless M-Bus etc., These protocols provide connectivity between M2M nodes within an M2M area network.

- The communication network provides connectivity to remote M2M area networks. The communication network provides connectivity to remote M2M area network. The communication network can use either wired or wireless network (IP based). While the M2M are networks use either properitorary or non-IP based communication protocols, the communication network uses IP-based network. Since non-IP based protocols are used within M2M area network, the M2M nodes within one network cannot communicate with nodes in an external network.
- To enable the communication between remote M2M are network, M2M gateways are used.

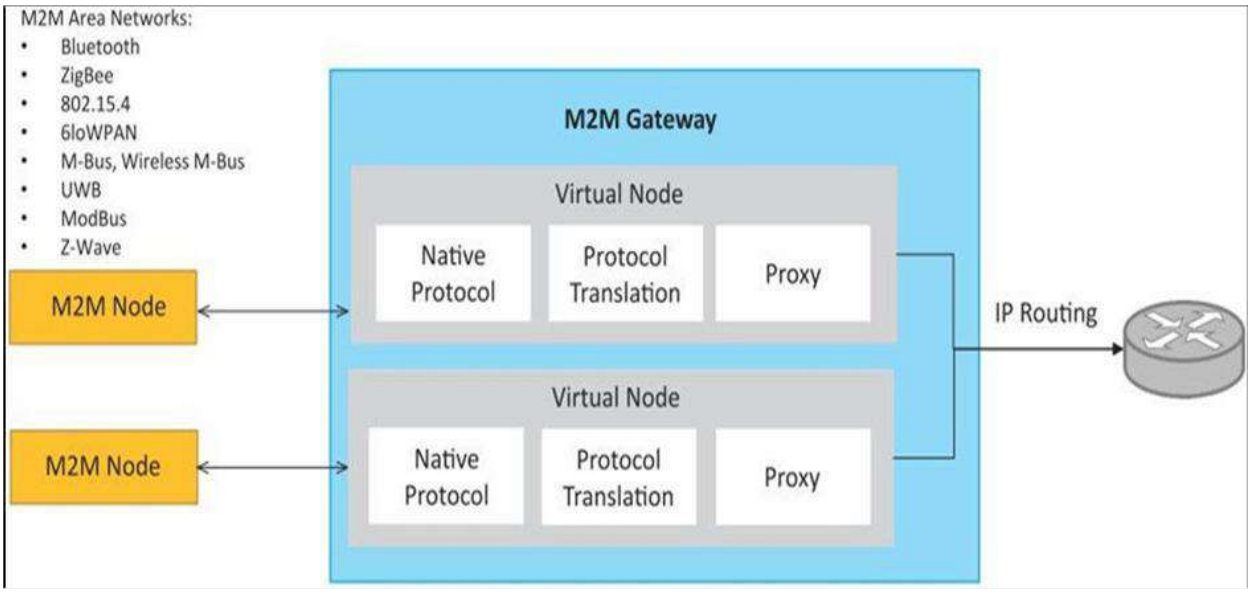


Fig. Shows a block diagram of an M2M gateway.

The communication between M2M nodes and the M2M gateway is based on the communication protocols which are naive to the M2M are network. M2M gateway performs protocol translations to enable Ip-connectivity for M2M are networks. M2M gateway acts as a proxy performing translations from/to native protocols to/from Internet Protocol(IP). With an M2M gateway, each mode in an M2M area network appears as a virtualized node for external M2M area networks.

Differences between IoT and M2M

1) Communication Protocols:

- Commonly uses M2M protocols include ZigBee, Bluetooth, ModBus, M-Bus, WirelessM-Bus etc.,
- In IoT uses HTTP, CoAP, WebSocket, MQTT,XMPP,DDS,AMQPetc.,

2) Machines in M2M Vs Things inIoT:

- Machines in M2M will be homogenous whereas Things in IoT will be heterogeneous.

3) Hardware Vs Software Emphasis:

- The emphasis of M2M is more on hardware with embedded modules, the emphasis of IoT is more on software.

4) Data Collection &Analysis

- M2M data is collected in point solutions and often in on-premises storage infrastructure.
- The data in IoT is collected in the cloud (can be public, private or hybrid cloud).

5) Applications

- M2M data is collected in point solutions and can be accessed by on-premises applications such as diagnosis applications, service management applications, and on- premisis enterprise applications.
- IoT data is collected in the cloud and can be accessed by cloud applications such as analytics applications, enterprise applications, remote diagnosis and management applications, etc.

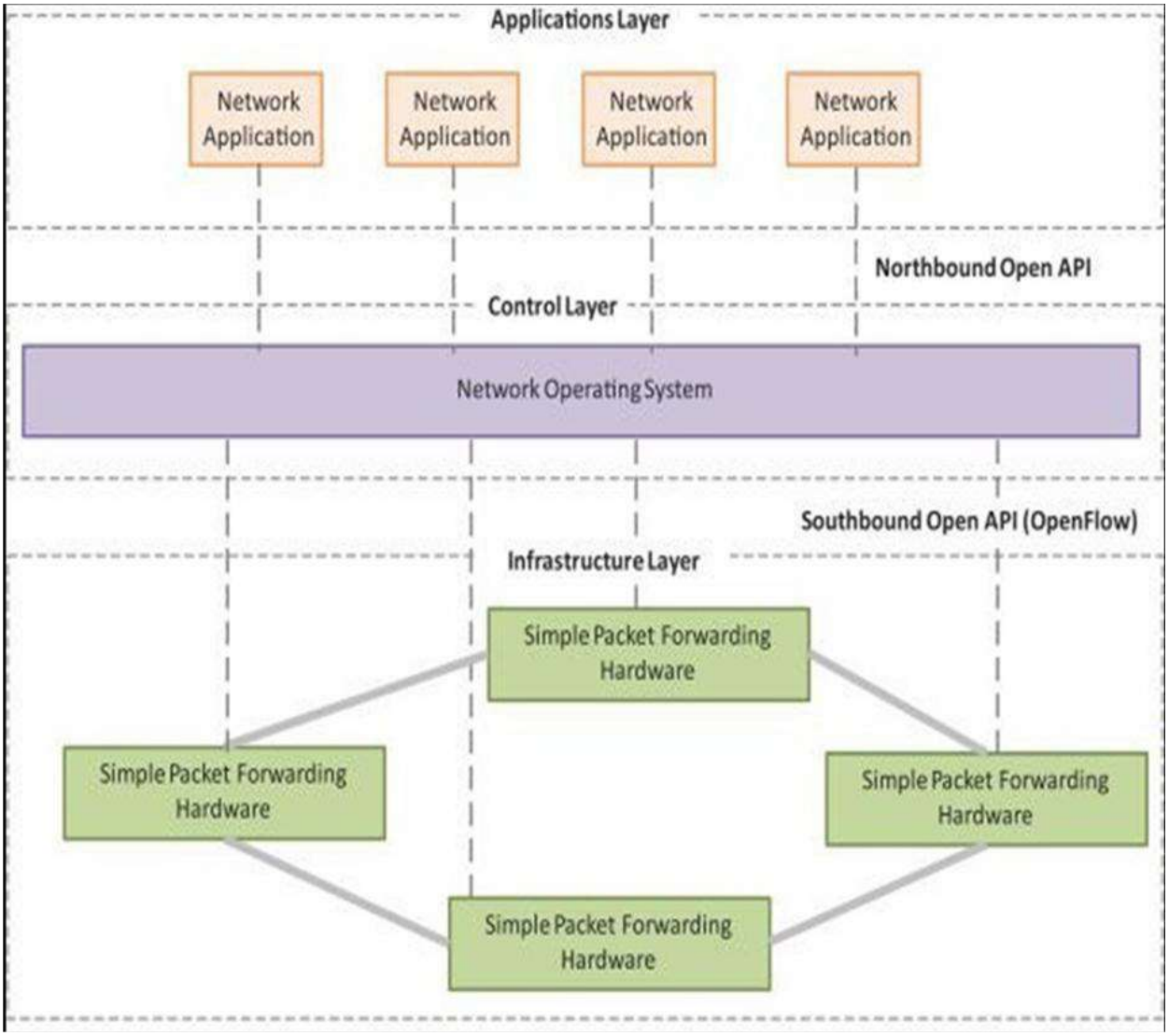
SDN and NVF for IoT

Software Defined Networking (SDN):

• Software-Defined Networking (SDN) is a networking architecture that separates the control plane from the data plane and centralizes the network controller.

• Software-based SDN controllers maintain a unified view of the network

• The underlying infrastructure in SDN uses simple packet forwarding hardware as opposed to specialized hardware in conventional networks.



SDN Architectur e

Key elements of SDN:

1) Centralized Network Controller

With decoupled control and data planes and centralized network controller, the network administrators can rapidly configure the network.

2) Programmable Open APIs

SDN architecture supports programmable open APIs for interface between the SDN application and control layers (Northbound interface).

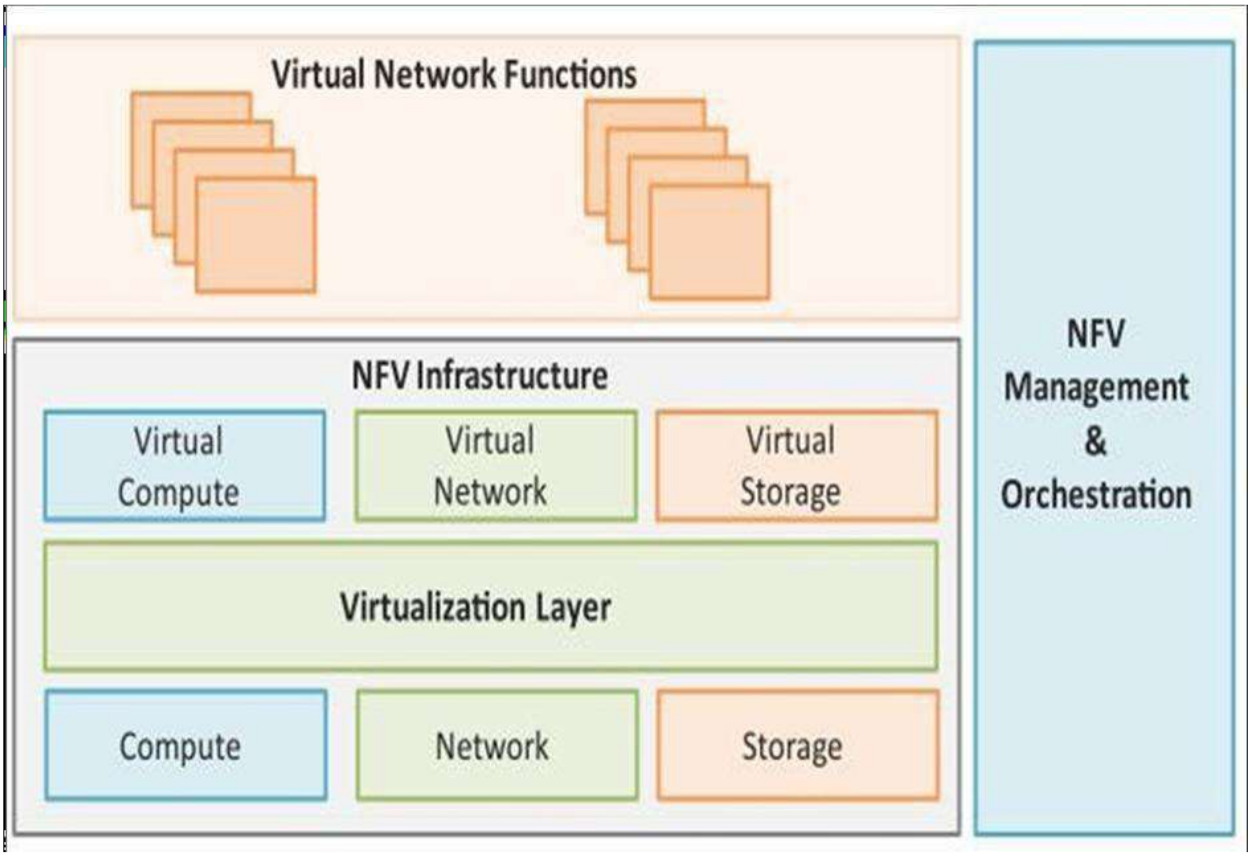
3) Standard Communication Interface (Open Flow)

SDN architecture uses a standard communication interface between the control and infrastructure layers (Southbound interface). Open Flow, which is defined by the Open Networking Foundation (ONF) is the broadly accepted SDN protocol for the South bound interface.

Network Function Virtualization (NFV)

• Network Function Virtualization (NFV) is a technology that leverages virtualization to consolidate the heterogeneous network devices onto industry standard high volume servers, switches and storage.

• NFV is complementary to SDN as NFV can provide the infrastructure on which SDN can run.



Key elements of NFV:

NFV Architecture

1) Virtualized Network Function (VNF):

VNF is a software implementation of a network function which is capable of running over the NFV Infrastructure (NFVI).

2) NFV Infrastructure (NFVI):

NFVI includes compute, network and storage resources that are virtualized.

3) NFV Management and Orchestration:

NFV Management and Orchestration focuses on all virtualization-specific management tasks and covers the orchestration and life-cycle management of physical and/or software resources that support the infrastructure virtualization, and the life-cycle management of VNFs.

Need for IoT Systems Management

Managing multiple devices within a single system requires advanced management capabilities.

1) **Automating Configuration:** IoT system management capabilities can help in automating the system configuration.

2) **Monitoring Operational & Statistical Data:** Management systems can help in monitoring opeartional and statistical data of a system. This data can be used for fault diagnosis or prognosis.

3) **Improved Reliability:** A management system that allows validating the system configurations before they are put into effect can help in improving the system reliability.

4) **System Wide Configurations:** For IoT systems that consists of multiple devices or nodes, ensuring system wide configuration can be critical for the correct functioning of the system.

5) **Multiple System Configurations:** For some systems it may be desirable to have multiple valid configurations which are applied at different times or in certain conditions.

6) **Retrieving & Reusing Configurations:** Management systems which have the capability of retrieving configurations from devices can help in reusing the configurations for other devices of the same type.

Simple Network Management Protocol (SNMP):

- SNMP is a well-known and widely used network management protocol that allows monitoring and configuring network devices such as routers, switches, servers, printers, etc.
- SNMP component include
- Network Management Station (NMS)
- Managed Device
- Management Information Base (MIB)
- SNMP Agent that runs on the device

Limitations of SNMP:

- SNMP is stateless in nature and each SNMP request contains all the information to process the request. The application needs to be intelligent to manage the device.
- SNMP is a connectionless protocol which uses UDP as the transport protocol, making it unreliable as there was no support for acknowledgement of requests.
- MIBs often lack writable objects without which device configuration is not possible using SNMP.

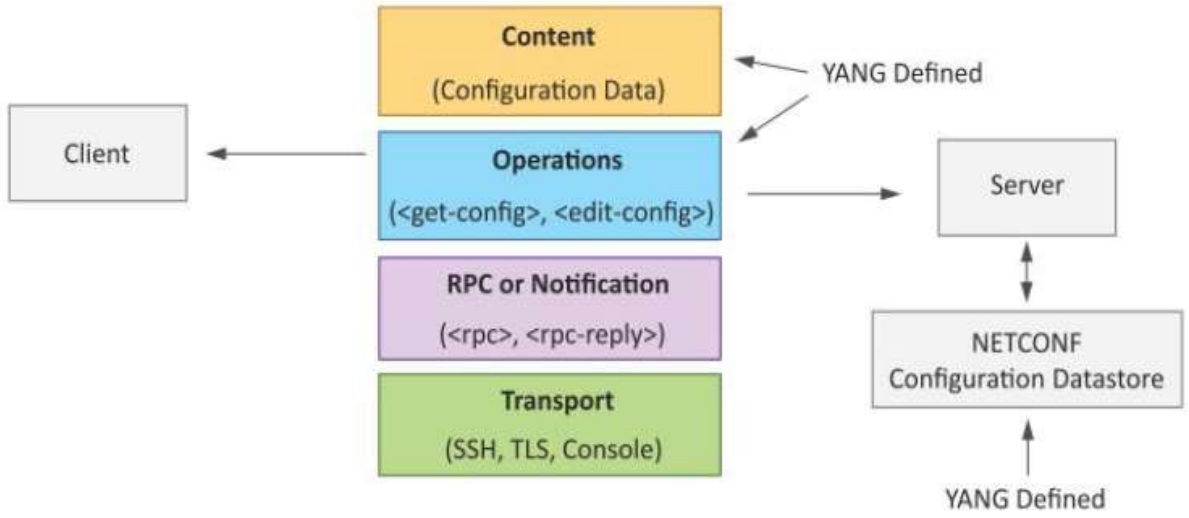
- It is difficult to differentiate between configuration and state data in MIBs.
- Retrieving the current configuration from a device can be difficult with SNMP.
- Earlier versions of SNMP did not have strong security features.

Network Operator Requirements:

- Ease of use
- Distinction between configuration and state data
- Fetch configuration and state data separately
- Configuration of the network as a whole
- Configuration transactions across devices
- Configuration deltas
- Dump and restore configurations
- Configuration validation
- Configuration database schemas
- Comparing configurations
- Role-based access control
- Consistency of access control lists:
- Multiple configuration sets
- Support for both data-oriented and task-oriented access control

NETCONF:

- Network Configuration Protocol (NETCONF) is a session-based network management protocol. NETCONF allows retrieving state or configuration data and manipulating configuration data on network devices



- NETCONF works on SSH transport protocol.
- Transport layer provides end-to-end connectivity and ensure reliable delivery of messages.
- NETCONF uses XML-encoded Remote Procedure Calls (RPCs) for framing request and response messages.
- The RPC layer provides mechanism for encoding of RPC calls and notifications.
- NETCONF provides various operations to retrieve and edit configuration data from network devices.
- The Content Layer consists of configuration and state data which is XML-encoded.
- The schema of the configuration and state data is defined in a data modeling language called YANG.
- NETCONF provides a clear separation of the configuration and state data.
- The configuration data resides within a NETCONF configuration data store on the server.

YANG:

- YANG is a data modeling language used to model configuration and state data manipulated by the NETCONF protocol
- YANG modules contain the definitions of the configuration data, state data, RPC calls that can be issued and the format of the notifications.

- YANG modules defines the data exchanged between the NETCONF client and server.
- A module comprises of a number of 'leaf' nodes which are organized into a hierarchical tree structure.
- The 'leaf' nodes are specified using the 'leaf' or 'leaf-list' constructs.
- Leaf nodes are organized using 'container' or 'list' constructs.
- A YANG module can import definitions from other modules.
- Constraints can be defined on the data nodes, e.g. allowed values.
- YANG can model both configuration data and state data using the 'config' statement.

YANG Module Example:

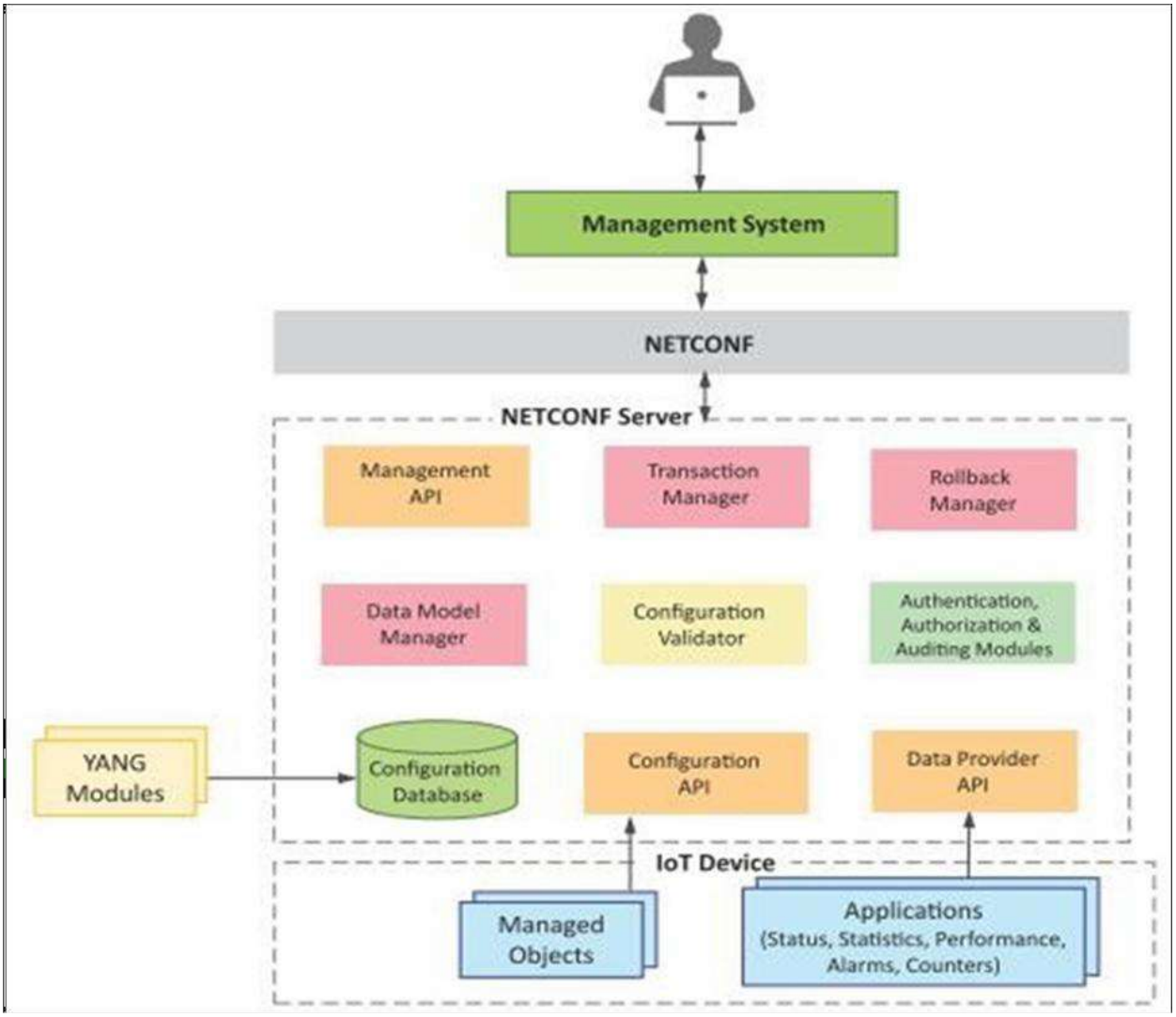
- This YANG module is a YANG version of the toaster MIB
- The toaster YANG module begins with the header information followed by identity declarations which define various bread types.
- The leaf nodes (‘toaster Manufacturer’, ‘toaster Model Number’ and ‘toaster Status’) are defined in the ‘toaster’ container.
- Each leaf node definition has a type and optionally a description and default value.
- The module has two RPC definitions (‘make-toast’ and ‘cancel-toast’).

IoT Systems Management with NETCONF-YANG

YANG is a data modeling language used to model configuration and state data manipulated by the NETCONF protocol.

The generic approach of IoT device management with NETCONF-YANG. Roles of various components are:

1) Management System
2) Management API
3) Transaction Manager
4) Rollback Manager
5) Data Model Manager
6) Configuration Validator
7) Configuration Database
8) Configuration API
9) Data Provider API



Management System: The operator uses a management system to send NETCONF messages to configure the IoT device and receives state information and notifications from the device as NETCONF messages.

2) **Management API:** allows management application to start NETCONF sessions.

3) **Transaction Manager:** executes all the NETCONF transactions and ensures that ACID properties hold true for the transactions.

4) **Rollback Manager:** is responsible for generating all the transactions necessary to rollback a current configuration to its original state.

5) **Data Model Manager:** Keeps track of all the YANG data models and the corresponding managed objects. Also keeps track of the applications which provide data for each part of a datam, odel.

6) **Configuration Validator:** checks if the resulting configuration after applying a transaction would be a valid configuration.

7) **Configuration Database:** contains both configuration and operational data.

8) **Configuration API:** Using the configuration API the application on the IoT device can be read configuration data from the configuration data store and write operational data to the operational data store.

9) **Data Provider API:** Applications on the IoT device can register for callbacks for various events using the Data Provider API. Through the Data Provider API, the applications can report statistics and operational data.

Steps for IoT device Management with NETCONF-YANG

1) Create a YANG model of the system that defines the configuration and state data of the system.

2) Complete the YANG model with the _Inc tool' which comes with Libnetconf.

3) Fill in the IoT device management code in the Trans API module.

4) Build the callbacks C file to generate the library file.

5) Load the YANG module and the TransAPI module into the Netopeer server using Netopeer manager tool.

6) The operator can now connect from the management system to the Netopeer server using the NetopeerCLI.

7) Operator can issue NETCONF commands from the Netopeer CLI. Command can be issued to change the configuration data, get operational data or execute an RPC on the IoT device.

UNIT - III

INTRODUCTION TO PYTHON:

Python is one of the most dynamic and versatile programming languages available in the industry today. Since its inception in the 1990s, Python has become hugely popular and even today there are thousands who are learning this Object-Oriented Programming language. If you are new to the world of programming, you have already heard the buzz it has created in recent times because of the features of Python and must be wondering what makes this programming language special.

Python is an object-oriented programming language that is designed in C. By nature, it is a high-level programming language that allows for the creation of both simple as well as complex operations. Along with this Python comes inbuilt with a wide array of modules as well as libraries which allows it to support many different programming languages like Java, C, C++, and JSON.

Python is a general-purpose high level programming language and suitable for providing a solid foundation to the reader in the area of cloud computing.

The main characteristics of Python are:

1) Multi-paradigm programming language.

2) Python supports more than one programming paradigms including object- oriented programming and structured programming.

3) Interpreted Language.

4) Python is an interpreted language and does not require an explicit compilation step.

5) The Python interpreter executes the program source code directly, statement by statement, as a processor or scripting engine does.

6) Interactive Language

7) Python provides an interactive mode in which the user can submit commands at the Python prompt and interact with the interpreter directly.

Features of Python

As a programming language, the features of Python brought to the table are many. Some of the most significant features of Python are:

Easy to Code: Python is a very developer-friendly language which means that anyone and everyone can learn to code it in a couple of hours or days. As compared to other object-oriented programming languages like Java, C, C++, and C#, Python is one of the easiest to learn.

Open Source and Free: Python is an open-source programming language which means that anyone can create and contribute to its development. Python has an online forum where thousands of coders gather daily to improve this language further. Along with this Python is free to download and use in any operating system, be it Windows, Mac or Linux.

Support for GUI: GUI or Graphical User Interface is one of the key aspects of any programming language because it has the ability to add flair to code and make the results more visual. Python has support for a wide array of GUIs which can easily be imported to the interpreter, thus making this one of the most favorite languages for developers.

Object-Oriented Approach: One of the key aspects of Python is its object-oriented approach. This basically means that Python recognizes the concept of class and object encapsulation thus allowing programs to be efficient in the long run.

High-Level Language: Python has been designed to be a high-level programming language, which means that when you code in Python you don't need to be aware of the coding structure, architecture as well as memory management.

Integrated by Nature: Python is an integrated language by nature. This means that the python interpreter executes codes one line at a time. Unlike other object-oriented programming languages, we don't need to compile Python code thus making the debugging process much easier and efficient. Another advantage of this is, that upon execution the Python code is immediately converted into an intermediate form also known as byte-code which makes it easier to execute and also saves runtime in the long run.

Highly Portable: Suppose you are running Python on Windows and you need to shift the same to either a Mac or a Linux system, then you can easily achieve the same in Python without having to worry about changing the code. This is not possible in other programming languages, thus making Python one of the most portable languages available in the industry.

Highly Dynamic: As mentioned in an earlier paragraph, Python is one of the most dynamic languages available in the industry today. What this basically means is that the type of a variable is decided at the run time and not in advance. Due to the presence of this feature, we do not need to specify the type of the variable during coding, thus saving time and increasing efficiency.

Extensive Array of Library: Out of the box, Python comes inbuilt with a large number of libraries that can be imported at any instance and be used in a specific program. The presence of libraries also makes sure that you don't need to write all the code yourself and can import the same from those that already exist in the libraries.

Support for Other Languages: Being coded in C, Python by default supports the execution of code written in other programming languages such as Java, C, and C#, thus making it one of the versatile in the industry.

Python Data Types:

Data types are the classification or categorization of data items. Data types represent a kind of value which determines what operations can be performed on that data. Numeric, non-numeric and Boolean (true/false) data are the most used data types. However, each programming language has its own classification largely reflecting its programming philosophy.

Python has the following standard or built-in data types: Built-in Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	Dict
Set Types:	set, frozenset
Boolean Type:	Bool
Binary Types:	bytes, bytearray, memoryview

Numeric

A numeric value is any representation of data which has a numeric value. Python identifies three types of numbers:

- **Integer:** Positive or negative whole numbers (without a fractional part)
- **Float:** Any real number with a floating point representation in which a fractional component is denoted by a decimal symbol or scientific notation
- **Complex number:** A number with a real and imaginary component represented as x+yj. x and y are floats and j is -1(square root of -1 called an imaginary number)

Boolean

Data with one of two built-in values `True` or `False`. Notice that 'T' and 'F' are capital. `true` and `false` are not valid booleans and Python will throw an error for them.

Sequence Type

A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types:

- **String**: A string value is a collection of one or more characters put in single, double or triple quotes.
- **List** : A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.
- **Tuple**: A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

Dictionary

A dictionary object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets. For example: `{1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}`

type() function

Python has an in-built function **type()** to ascertain the data type of a certain value. For example, enter `type(1234)` in Python shell and it will return `<class 'int'>`, which means 1234 is an integer value. Try and verify the data type of different values in Python shell, as shown below.

Mutable and Immutable Objects

Data objects of the above types are stored in a computer's memory for processing. Some of these values can be modified during processing, but contents of others can't be altered once they are created in the memory.

Number values, strings, and tuple are immutable, which means their contents can't be altered after creation.

On the other hand, collection of items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are mutable objects.

types of Data Structures in Python

Python has **implicit** support for Data Structures which enable you to store and access data. These structures are called List, Dictionary, Tuple and Set.

Python allows its users to create their own Data Structures enabling them to have **full control** over their functionality. The most prominent Data Structures are Stack, Queue, Tree, Linked List and so on which are also available to you in other programming languages. So now that you know what are the types available to you, why don't we move ahead to the Data Structures and implement them using Python.

Built-in Data Structures

As the name suggests, these Data Structures are built-in with Python which makes programming easier and helps programmers use them to obtain solutions faster. Let's discuss each of them in detail.

Lists

Lists are used to store data of different data types in a sequential manner. There are addresses assigned to every element of the list, which is called as Index. The index value starts from 0 and goes on until the last element called the **positive index**. There is also **negative indexing** which starts from -1 enabling you to access elements from the last to first. Let us now understand lists better with the help of an example program.

Creating a list

To create a list, you use the square brackets and add elements into it accordingly. If you do not pass any elements inside the square brackets, you get an empty list as the output.

```
1my_list = [] #create empty list

2print(my_list)

3my_list = [1, 2, 3, 'example', 3.132] #creating list with data

4print(my_list)
```

Output:

```
[]
[1, 2, 3, 'example', 3.132]
```

Dictionary

Dictionaries are used to store **key-value** pairs. To understand better, think of a phone directory where hundreds and thousands of names and their corresponding numbers have been added. Now the constant values here are Name and the Phone Numbers which are called as the keys. And the various names and phone numbers are the values that have been fed to the keys. If you access the values of the keys, you will obtain all the names and phone numbers. So that is what a key-value pair is. And in Python, this structure is stored using Dictionaries. Let us understand this better with an example program.

Creating a Dictionary

Dictionaries can be created using the flower braces or using the dict() function. You need to add the key-value pairs whenever you work with dictionaries.

```
1my_dict = {} #empty dictionary

2print(my_dict)

3my_dict = {1: 'Python', 2: 'Java'} #dictionary with elements

4print(my_dict)
```

Output:

```
{}
{1: ‘Python’, 2: ‘Java’}
```

Tuple

Tuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what. The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed. The example program will help you understand better.

Creating a Tuple

You create a tuple using parenthesis or using the tuple() function.

```
1my_tuple = (1, 2, 3) #create tuple
2print(my_tuple)
```

Output:
(1, 2, 3)

Sets

Sets are a collection of unordered elements that are unique. Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic. The operations also are the same as is with the arithmetic sets. An example program would help you understand better.

Creating a set

Sets are created using the flower braces but instead of adding key-value pairs, you just pass values to it.

```
1my_set = {1, 2, 3, 4, 5, 5, 5} #create set
2print(my_set)
```

Output:
{1, 2, 3, 4, 5}

User-Defined Data Structures

Arrays vs. Lists

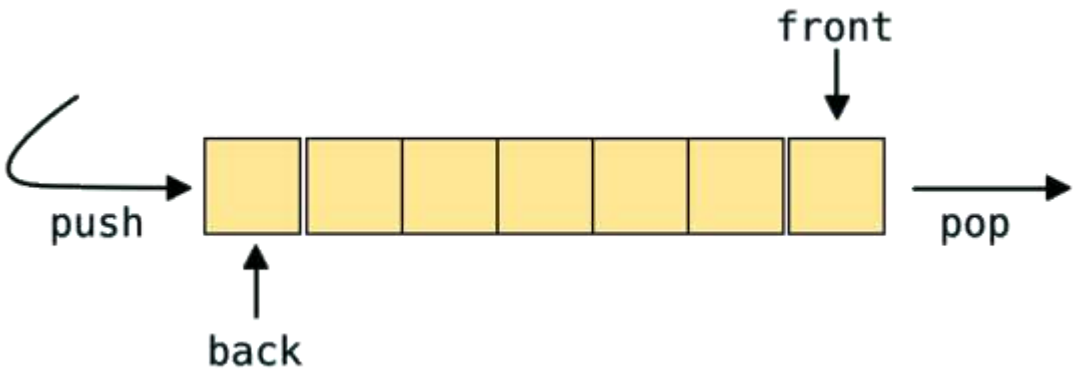
Arrays and lists are the same structure with one difference. Lists allow heterogeneous data element storage whereas Arrays allow only homogenous elements to be stored within them.

Stack

Stacks are linear Data Structures which are based on the principle of Last-In-First-Out (LIFO) where data which is entered last will be the first to get accessed. It is built using the array structure and has operations namely, pushing (adding) elements, popping (deleting) elements and accessing elements only from one point in the stack called as the TOP. This TOP is the pointer to the current position of the stack. Stacks are prominently used in applications such as Recursive Programming, reversing words, undo mechanisms in word editors and so forth.

Queue

A queue is also a linear data structure which is based on the principle of First-In-First-Out (FIFO) where the data entered first will be accessed first. It is built using the array structure and has operations which can be performed from both ends of the Queue, that is, head-tail or front-back. Operations such as adding and deleting elements are called En-Queue and De-Queue and accessing the elements can be performed. Queues are used as Network Buffers for traffic congestion management, used in Operating Systems for Job Scheduling and many more.

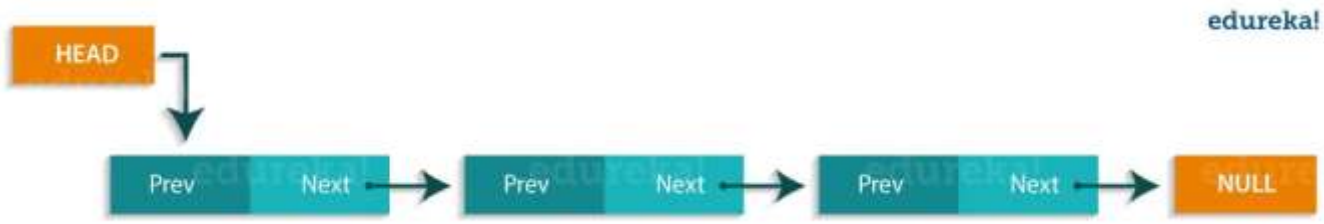


Tree

Trees are non-linear Data Structures which have root and nodes. The root is the node from where the data originates and the nodes are the other data points that are available to us. The node that precedes is the parent and the node after is called the child. There are levels a tree has to show the depth of information. The last nodes are called the leaves. Trees create a hierarchy which can be used in a lot of real-world applications such as the HTML pages use trees to distinguish which tag comes under which block. It is also efficient in searching purposes and much more.

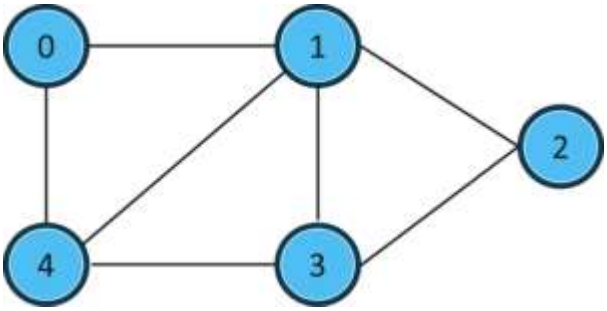
Linked List

Linked lists are linear Data Structures which are not stored consequently but are linked with each other using pointers. The node of a linked list is composed of data and a pointer called next. These structures are most widely used in image viewing applications, music player applications and so forth.



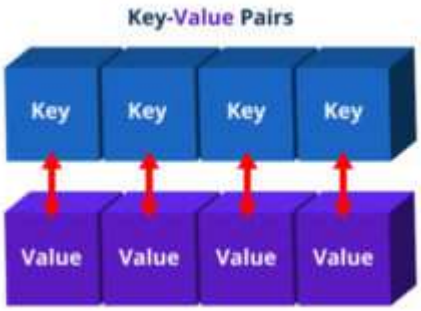
Graph

Graphs are used to store data collection of points called vertices (nodes) and edges (edges). Graphs can be called as the most accurate representation of a real-world map. They are used to find the various cost-to-distance between the various data points called as the nodes and hence find the least path. Many applications such as Google Maps, Uber, and many more use Graphs to find the least distance and increase profits in the best ways.



HashMaps

HashMaps are the same as what dictionaries are in Python. They can be used to implement applications such as phonebooks, populate data according to the lists and much more.



Control Flow Tools:

Besides the while statement just introduced, Python uses the usual flow control statements known from other languages, with some twists.

1. if Statements

Perhaps the most well-known statement type is the if statement. For example:

```
>>>
```

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

There can be zero or more elif parts, and the else part is optional. The keyword 'elif' is short for 'else if', and is useful to avoid excessive indentation. An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

2. for Statements

The for statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>>
```

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

Code that modifies a collection while iterating over that same collection can be tricky to get right. Instead, it is usually more straight-forward to loop over a copy of the collection or to create a new collection:

```
# Strategy:  Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy:  Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

3. The range() Function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions:

```
>>>
```

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

The given end point is never part of the generated sequence; range(10) generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
range(5, 10)
   5, 6, 7, 8, 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

To iterate over the indices of a sequence, you can combine range() and len() as follows:

```
>>>
```

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

In most such cases, however, it is convenient to use the enumerate() function, see Looping Techniques.

A strange thing happens if you just print a range:

```
>>>
```

```
>>> print(range(10))
range(0, 10)
```

In many ways the object returned by range() behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the for statement is such a construct, while an example of a function that takes an iterable is sum():

```
>>>
```

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Later we will see more functions that return iterables and take iterables as arguments. Lastly, maybe you are curious about how to get a list from a range. Here is the solution:

```
>>>
```

```
>>> list(range(4))
[0, 1, 2, 3]
```

In chapter Data Structures, we will discuss in more detail about list().

4. break and continue Statements, and else Clauses on Loops

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the iterable (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>>
```

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the else clause belongs to the for loop, **not** the if statement.)

When used with a loop, the else clause has more in common with the else clause of a try statement than it does with that of if statements: a try statement's else clause runs when no exception occurs, and a loop's else clause runs when no break occurs. For more on the try statement and exceptions, see Handling Exceptions.

The continue statement, also borrowed from C, continues with the next iteration of the loop:

```
>>>
```

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
```

Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9

5. pass Statements

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
>>>
```

```
>>> while True:
...     pass  # Busy-wait for keyboard interrupt (Ctrl+C)
...
```

This is commonly used for creating minimal classes:

```
>>>
```

```
>>> class MyEmptyClass:
...     pass
...
```

Another place pass can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The pass is silently ignored:

```
>>>
```

```
>>> def initlog(*args):
...     pass   # Remember to implement this!
...
```

Functions:

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
>>>
```

```
>>> def fib(n):    # write Fibonacci series up to n
```

```
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword def introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can be found in the section Documentation Strings) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a global statement, or, for variables of enclosing functions, named in a nonlocal statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object). When a function calls another function, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that fib is not a function but a procedure since it doesn't return a value. In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called None (it's a built-in name). Writing the value None is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using print():

```
>>> fib(0)
>>> print(fib(0))
None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
>>> def fib2(n):  # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The return statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.

- The statement result.append(a) calls a *method* of the list object result. A method is a function that 'belongs' to an object and is named obj.methodname, where obj is some object (this may be an expression), and methodname is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see Classes) The method append() shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to result = result + [a], but more efficient.

4.7. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

4.7.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')
        print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: ask_ok('Do you really want to quit?')
- giving one of the optional arguments: ask_ok('OK to overwrite the file?', 2)
- or even giving all arguments: ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')

This example also introduces the in keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

This will print

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
def f(a, L=None):
    if L is None:
        L = []
```

```
    L.append(a)
    return L
```

2. Keyword Arguments

Functions can also be called using keyword arguments of the form kwarg=value. For instance, the following function:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

accepts one required argument (voltage) and three optional arguments (state, action, and type). This function can be called in any of the following ways:

```
parrot(1000)                                          # 1 positional argument
parrot(voltage=1000)                                  # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')             # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)             # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump')         # 3 positional arguments
parrot('a thousand', state='pushing up the daisies')  # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument after a keyword argument
parrot(110, voltage=220)     # duplicate value for the same argument
parrot(actor='John Cleese')  # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actor is not a valid argument for the parrot function), and their order is not important. This also includes non-optional arguments (e.g. parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
>>>
```

```
>>> def function(a):
```

```
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form **name is present, it receives a dictionary (see Mapping Types — dict) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form *name (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (*name must occur before **name.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
shopkeeper : Michael Palin
```

client : John Cleese
sketch : Cheese Shop Sketch

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |          |                |
        |       Positional or keyword   |
        |                              - Keyword only
         -- Positional only
```

where / and * are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters.

3.1. Positional-or-Keyword Arguments

If / and * are not present in the function definition, arguments may be passed to a function by position or by keyword.

3.2. Positional-Only Parameters

Looking at this in a bit more detail, it is possible to mark certain parameters as *positional-only*. If *positional-only*, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a / (forward-slash). The / is used to logically separate the positional-only parameters from the rest of the parameters. If there is no / in the function definition, there are no positional-only parameters.

Parameters following the / may be *positional-or-keyword* or *keyword-only*.

3.3. Keyword-Only Arguments

To mark parameters as *keyword-only*, indicating the parameters must be passed by keyword argument, place an * in the arguments list just before the first *keyword-only* parameter.

3.4. Function Examples

Consider the following example function definitions paying close attention to the markers / and *:

```
>>>
```

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

The first function definition, standard_arg, the most familiar form, places no restrictions on the calling convention and arguments may be passed by position or keyword:

```
>>>
```

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
```

Modules and Packages:

In programming, a module is a piece of software that has a specific functionality. For example, when building a ping pong game, one module would be responsible for the game logic,

and another module would be responsible for drawing the game on the screen. Each module is a different file, which can be edited separately.

Writing modules

Modules in Python are simply Python files with a .py extension. The name of the module will be the name of the file. A Python module can have a set of functions, classes or variables defined and implemented.

The Python script `game.py` will implement the game. It will use the function `draw_game` from the file `draw.py`, or in other words, the`draw` module, that implements the logic for drawing the game on the screen.

Importing module objects to the current namespace

You may have noticed that in this example, `draw_game` does not precede with the name of the module it is imported from, because we've specified the module name in the `import` command.

The advantages of using this notation is that it is easier to use the functions inside the current module because you don't need to specify which module the function comes from. However, any namespace cannot have two objects with the exact same name, so the `import` command may replace an existing object in the namespace.

Importing all objects from a module

This might be a bit risky as changes in the module might affect the module which imports it, but it is shorter and also does not require you to specify which objects you wish to import from the module.

Custom import name

We may also load modules under any name we want. This is useful when we want to import a module conditionally to use the same name in the rest of the code.

Module initialization

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

This is useful to know, because this means that you can rely on this behavior for initializing objects.

Extending module load path

There are a couple of ways we could tell the Python interpreter where to look for modules, aside from the default, which is the local directory and the built-in modules. You could either use the environment variable `PYTHONPATH` to specify additional directories to look for modules.

This will execute `game.py`, and will enable the script to load modules from the `foo` directory as well as the local directory.

Another method is the `sys.path.append` function.

Exploring built-in modules

Check out the full list of built-in modules in the Python standard library here.

Two very important functions come in handy when exploring modules in Python - the `dir` and `help` functions.

If we want to import the module `urllib`, which enables us to create read data from URLs.

When we find the function in the module we want to use, we can read about it more using the `help` function, inside the Python interpreter:

- script.py
- IPython Shell

Writing packages:

Packages are namespaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which **MUST** contain a special file called `__init__.py`. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called `foo`, which marks the package name, we can then create a module inside that package called `bar`. We also must not forget to add the `__init__.py` file inside the `foo` directory.

To use the module `bar`, we can import it in two ways:

- script.py
- IPython Shell.

UNIT – IV
IoT Physical Devices and Endpoints

IoT Device

A "Thing" in Internet of Things (IoT) can be any object that has a unique identifier and which can send/receive data (including user data) over a network (e.g., smart phone, smart TV, computer, refrigerator, car, etc.).

• IoT devices are connected to the Internet and send information about themselves or about their surroundings (e.g. information sensed by the connected sensors) over a network (to other devices or servers/storage) or allow actuation upon the physical entities/environment around them remotely.

IoT Device Examples

A home automation device that allows remotely monitoring the status of appliances and controlling the appliances.

• An industrial machine which sends information a bouts its operation and health monitoring data to a server.

• A car which sends information about its location to a cloud-based service.

• A wireless-enabled wearable device that measures data about a person such as the number of steps walked and sends the data to a cloud-based service.

Basic building blocks of an IoT Device

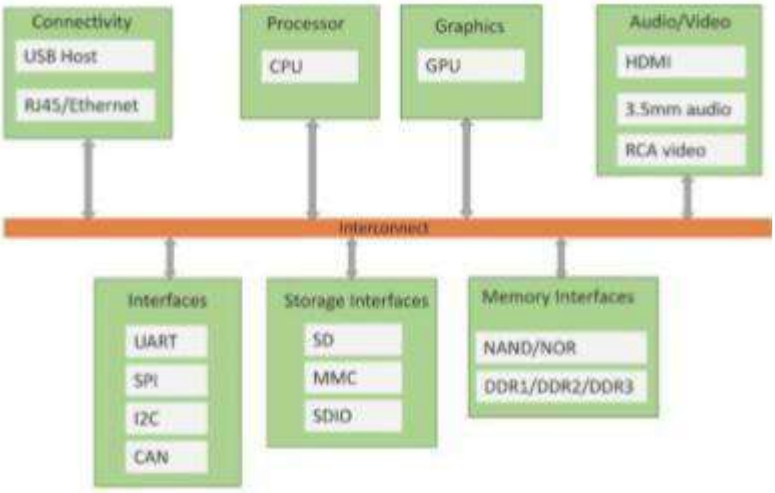
1. **Sensing:** Sensors can be either on-board the IoT device or attached to thedevice.

2. **Actuation:** IoT devices can have various types of actuators attached that allow taking actions upon the physical entities in the vicinity of thedevice.

3. **Communication:** Communication modules are responsible for sending collected data to other devices or cloud-based servers/storage and receiving data from other devices and commands from remote applications.

4. **Analysis & Processing:** Analysis and processing modules are responsible for making sense of the collecteddata.

Block diagram of an IoT Device



Exemplary Device: Raspberry Pi

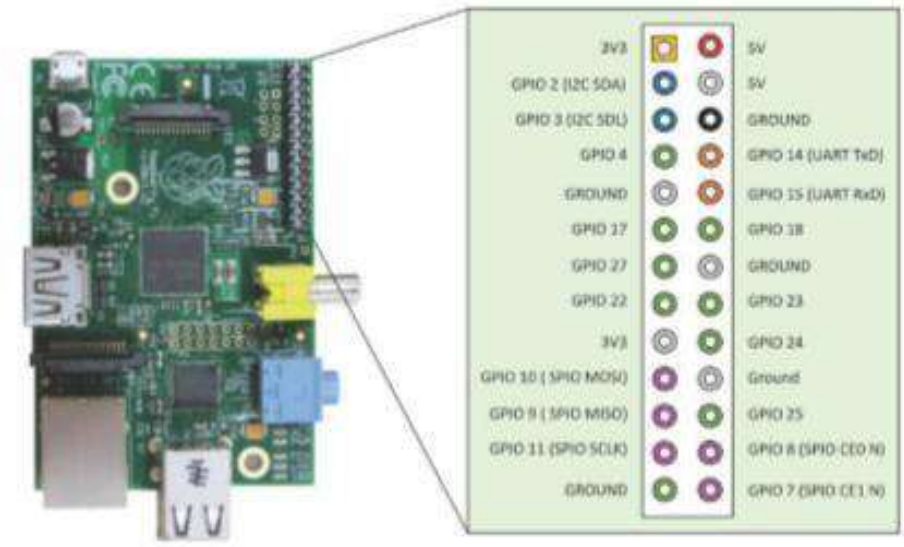
Raspberry Pi is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do. Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box". Raspberry Pi is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do. Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins. Since Raspberry Pi runs Linux operating system, it supports Python "out of the box".

Raspberry Pi

Linux on Raspberry Pi

1. Raspbian: Raspbian Linux is a Debian Wheezy port optimized for RaspberryPi.
2. Arch: Arch is an Arch Linux port for AMDdevices.
3. Pidora: Pidora Linux is a Fedora Linux optimized for RaspberryPi.
4. RaspBMC: RaspBMC is an XBMC media-center distribution for RaspberryPi.
5. OpenELEC: OpenELEC is a fast and user-friendly XBMC media-centerdistribution.
6. RISC OS: RISC OS is a very fast and compact operatingsystem.

Raspberry Pi GPIO



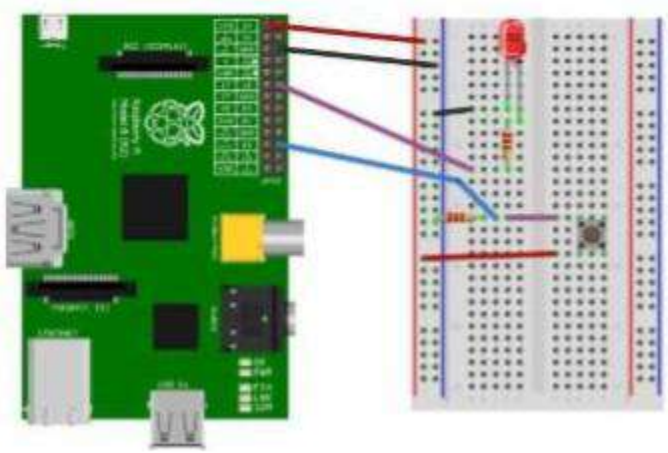
Raspberry Pi Interfaces

1. **Serial**: The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serialperipherals.

2. **SPI:** Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheraldevices.

3. **I2C:** The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clockline).

Raspberry Pi Example: Interfacing LED and switch with Raspberry Pi



from time import sleeP import RPi.GPIO asGPIO GPIO.setmode(GPIO.BCM)
#Switch Pin GPIO.setup(25,GPIO.IN) #LEDPin
GPIO.setup(18,GPIO.OUT)
state=false
deftoggleLED(pin):
state = not state GPIO.output(pin,state)
whileTrue:
try:
if (GPIO.input(25) ==True):
toggleLED(pin)
sleep(.01)
exceptKeyboardInterrupt:
exit()

Other Devices

1. pcDuino
2. BeagleBoneBlack
3. Cubieboard

UNIT V

IoT PHYSICAL SERVERS AND CLOUD OFFERINGS

Introduction to Cloud Computing

The Internet of Things (IoT) involves the internet-connected devices we use to perform the processes and services that support our way of life. Another component set to help IoT succeed is cloud computing, which acts as a sort of front end. Cloud computing is an increasingly popular service that offers several advantages to IOT, and is based on the concept of allowing users to perform normal computing tasks using services delivered entirely over the internet. A worker may need to finish a major project that must be submitted to a manager, but perhaps they encounter problems with memory or space constraints on their computing device. Memory and space constraints can be minimized if an application is instead hosted on the internet. The worker can use a cloud computing service to finish their work because the data is managed remotely by a server. Another example: you have a problem with your mobile device and you need to reformat it or reinstall the operating system. You can use Google Photos to upload your photos to internet-based storage. After the reformat or reinstall, you can then either move the photos back to you device or you can view the photos on your device from the internet when you want.

Concept

In truth, cloud computing and IoT are tightly coupled. The growth of IoT and the rapid development of associated technologies create a widespread connection of ―things. This has lead to the production of large amounts of data, which needs to be stored, processed and accessed. Cloud computing as a paradigm for big data storage and analytics. While IoT is exciting on its own, the real innovation will come from combining it with cloud computing. The combination of cloud computing and IoT will enable new monitoring services and powerful processing of sensory data streams. For example, sensory data can be uploaded and stored with cloud computing, later to be used intelligently for smart monitoring and actuation with other smart devices. Ultimately, the goal is to be able to transform data to insight and drive productive, cost-effective action from those insights. The cloud effectively serves as the brain to improved decision-making and optimized internet-based interactions. However, when IoT meets cloud, new challenges arise. There is an urgent need for novel network architectures that seamlessly integrate them. The critical concerns during integration are quality of service (QoS) and quality of experience (QoE), as well as data security, privacy and reliability. The virtual infrastructure for practical mobile computing and interfacing includes integrating applications, storage devices, monitoring devices, visualization platforms, analytics tools and client delivery. Cloud computing offers a practical utility-based model that will enable businesses and users to access applications on demand anytime and from anywhere

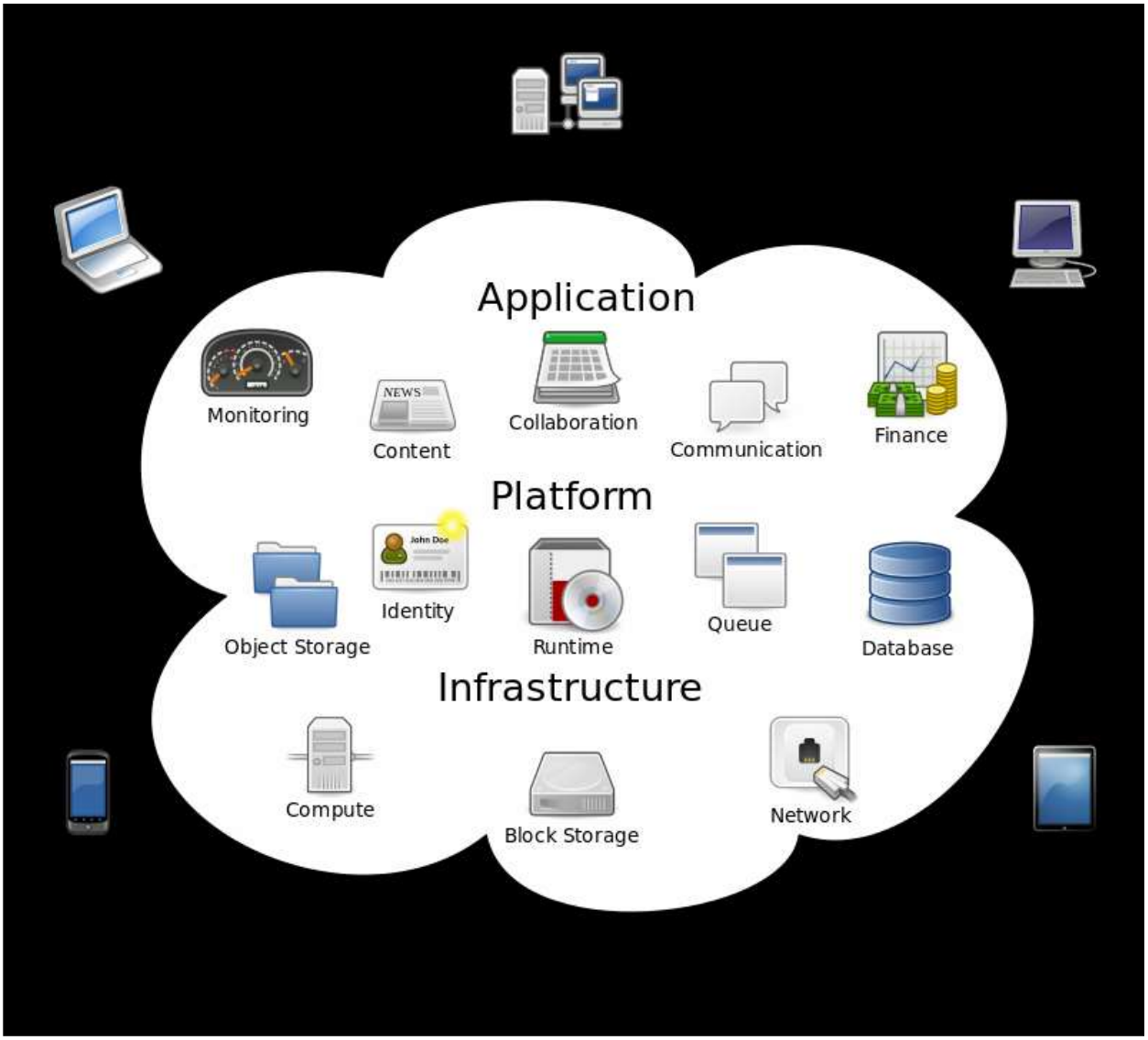


Fig: CLOUD CMPUTING

Characteristics

First, the cloud computing of IoT is an on-demand self service, meaning it‘s there when you need it. Cloud computing is a web-based service that can be accessed without any special assistance or permission from other people; however, you need at minimum some sort of internet access.

Second, the cloud computing of IoT involves broad network access, meaning it offers several connectivity options. Cloud computing resources can be accessed through a wide variety of internet-connected devices such as tablets, mobile devices and laptops. This level of

convenience means users can access those resources in a wide variety of manners, even from older devices. Again, though, this emphasizes the need for network access points.

Third, cloud computing allows for resource pooling, meaning information can be shared with those who know where and how (have permission) to access the resource, anytime and anywhere. This lends to broader collaboration or closer connections with other users. From an IoT perspective, just as we can easily assign an IP address to every "thing" on theplanet, we can share the "address" of the cloud-based protected and stored information with others and pool resources.

Fourth, cloud computing features rapid elasticity, meaning users can readily scale the service to their needs. You can easily and quickly edit your software setup, add or remove users, increase storage space, etc. This characteristic will further empower IoT by providing elastic computing power, storage and networking.

Finally, the cloud computing of IoT is a measured service, meaning you get what you pay for. Providers can easily measure usage statistics such as storage, processing, bandwidth and active user accounts inside your cloud instance. This pay per use (PPU) model means your costs scale with your usage. In IoT terms, it's comparable to the ever-growing network of physical objects that feature an IP address for internet connectivity, and the communication that occurs between these objects and other internet-enabled devices and systems; just like your cloud service, the service rates for that IoT infrastructure may also scale with use.

Service and Deployment Service models

Service delivery in cloud computing comprises three different service models: software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS).

Software as a service (SaaS) provides applications to the cloud‘s end user that are mainly accessed via a web portal or service-oriented architecture-based web service technology. These services can be seen as ASP (application service provider) on the application layer. Usually, a specific company that uses the service would run, maintain and give support so that it can be reliably used over a long period of time.

Platform as a service (PaaS) consists of the actual environment for developing and provisioning cloud applications. The main users of this layer are developers that want to develop and run a cloud application for a particular purpose. A proprietary language was supported and provided by the platform (a set of important basic services) to ease communication, monitoring, billing and other aspects such as startup as well as to ensure an application‘s scalability and flexibility. Limitations regarding the programming languages supported, the programming model, the ability to access resources, and the long-term persistence are possibledisadvantages.

Infrastructure as a service (IaaS) provides the necessary hardware and software upon which a customer can build a customized computing environment. Computing resources, data storage resources and the communications channel are linked together with these essential IT resources to ensure the stability of applications being used on the cloud. Those stack models can be

referred to as the medium for IoT, being used and conveyed by the users in different methods for the greatest chance of interoperability. This includes connecting cars, wearable's, TVs, smart phones, fitness equipment, robots, ATMs, and vending machines as well as the vertical applications, security and professional services, and analytics platforms that come with them.

Deployment models

Deployment in cloud computing comprises four deployment models: private cloud, public cloud, community cloud and hybrid cloud.

A private cloud has infrastructure that's provisioned for exclusive use by a single organization comprising multiple consumers such as business units. It may be owned, managed and operated by the organization, a third party or some combination of them, and it may exist on or off premises.

A public cloud is created for open use by the general public. Public cloud sells services to anyone on the internet. (Amazon Web Services is an example of a large public cloud provider.) This model is suitable for business requirements that require management of load spikes and the applications used by the business, activities that would otherwise require greater investment in infrastructure for the business. As such, public cloud also helps reduce capital expenditure and bring down operational ITcosts.

A community cloud is managed and used by a particular group or organizations that have shared interests, such as specific security requirements or a common mission.

Finally, a hybrid cloud combines two or more distinct private, community or public cloud infrastructures such that they remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability. Normally, information that's not critical is outsourced to the public cloud, while business-critical services and data are kept within the control of the organization.

CLOUD STORAGE API

A cloud storage API is an application program interface that connects a locally-based application to a cloud-based storage system, so that a user can send data to it and access and work with data stored in it. To the application, the cloud storage system is just another target device, like tape or disk-based storage. An application program interface (API) is code that allows two software programs to communicate with each other. The API defines the correct way for a developer to write a program that requests services from an operating system (OS) or other application. APIs are implemented by function calls composed of verbs and nouns. The required syntax is described in the documentation of the application beingcalled.

How APIs work

APIs are made up of two related elements. The first is a specification that describes how information is exchanged between programs, done in the form of a request for processing and a return of the necessary data. The second is a software interface written to that specification and published in some way for use.The software that wants to access the features and capabilities of the API is said to call it, and the software that creates the API is said to publish it.

Why APIs are important for business

The web, software designed exchange information via the internet and cloud computing have all combined to increase the interest in APIs in general and services in particular. Software that was once custom-developed for a specific purpose is now often written referencing APIs that provide broadly useful features, reducing development time and cost and mitigating the risk of errors.APIs have steadily improved software quality over the last decade, and the growing number of web services exposed through APIs by cloud providers is also encouraging the creation of cloud-specific applications, internet of things (IoT) efforts and apps to support mobile devices and users.

Three basic types of APIs

APIs take three basic forms: local, web-like and program-like.

1. **Local APIs** are the original form, from which the name came. They offer OS or middleware services to application programs. Microsoft's .NET APIs, the TAPI (Telephony API) for voice applications, and database access APIs are examples of the local APIform.

2. **Web APIs** are designed to represent widely used resources like HTML pages and are accessed using a simple HTTP protocol. Any web URL activates a web API. Web APIs are often called REST (representational state transfer) or RESTful because the publisher of REST interfaces doesn't save any data internally between requests. As such, requests from many users can be intermingled as they would be on theinternet.

3. **Program APIs** are based on remote procedure call (RPC) technology that makes a remote program component appear to be local to the rest of the software. Service oriented architecture (SOA) APIs, such as Microsoft's WS-series of APIs, are programAPIs.

IoT / Cloud Convergence

Internet-of-Things can benefit from the scalability, performance and pay-as-you-go nature of cloud computing infrastructures. Indeed, as IoT applications produce large volumes of data and comprise multiple computational components (e.g., data processing and analytics algorithms), their integration with cloud computing infrastructures could provide them with opportunities for cost-effective on-demand scaling. As prominent examples consider the following settings:

A Small Medium Enterprise (SME) developing an energy management IoT product, targeting smart homes and smart buildings. By streaming the data of the product (e.g., sensors and WSN data) into the cloud it can accommodate its growth needs in a scalable and cost effective fashion. As the SMEs acquires more customers and performs more deployments of its product, it is able tocollectandmanagegrowingvolumesofdatainascalableway,thustakingadvantageofa ―pay-as-you-grow‖model. Moreover, cloud integration allows the SME to store and processmassive datasets collected from multiple (rather than a single)deployments.

A smart city can benefit from the cloud-based deployment of its IoT systems and applications. A city is likely to deploy many IoT applications, such as applications for smart energy management, smart water management, smart transport management, urban mobility of the citizensandmore.Theseapplicationscomprisemultiplesensorsanddevices, along with computational components. Furthermore, they are likely to produce very large data volumes. Cloud integration enables the city to host these data and applications in a cost-effective way. Furthermore, the elasticity of the cloud can directly support expansions to these applications, but also the rapid deployment of new ones without major concerns about the provisioning of the required cloud computing resources.

A cloud computing provider offering pubic cloud services can extend them to the IoT area, through enabling third-parties to access its infrastructure in order to integrate IoT data and/or computational components operating over IoT devices. The provider can offer IoT data access and services in a pay-as-you-fashion, through enabling third-parties to access resources of its infrastructure and accordingly to charge them in a utility-based fashion.

These motivating examples illustrate the merit and need for converging IoT and cloud computing infrastructure. Despite these merits, this convergence has always been challenging mainly due to the conflicting properties of IoT and cloud infrastructures, in particular, IoT devices tend to be location specific, resource constrained, expensive (in terms of development/ deployment cost) and generally inflexible (in terms of resource access and availability). On the other hand, cloud computing resources are typically location independent and inexpensive, while at the same time providing rapid and flexibly elasticity. In order to alleviate these incompatibilities, sensors and devices are virtualized prior to integrating their data and services in the cloud, in order to enable their distribution across any cloud resources. Furthermore, service and sensor discovery functionalities are implementing on the cloud in order to enable the discovery of services and sensors that reside in different locations.

Based on these principles the IoT/cloud convergence efforts have started since over a decade i.e. since they very early days of IoT and cloud computing. Early efforts in the research community (i.e. during 2005-2009) have focused on streaming sensor and WSN data in a cloud infrastructure. Since 2007 we have also witnessed the emergence of public IoT clouds, including commercial efforts. One of the earliest efforts has been the famous Pachube.com infrastructure (used extensively for radiation detection and production of radiation maps during earthquakes in

Japan). Pachube.com has evolved (following several evolutions and acquisitions of this infrastructure) to Xively.com, which is nowadays one of the most prominent public IoT clouds. Nevertheless, there are tens of other public IoT clouds as well, such as ThingsWorx, ThingsSpeak, Sensor-Cloud, Realtime.io and more. The list is certainly non- exhaustive. These public IoT clouds offer commercial pay-as-you-go access to end-users wishing to deploying IoT applications on the cloud. Most of them come with developer friendly tools, which enable the development of cloud applications, thus acting like a PaaS for IoT in the cloud. Similarly to cloud computing infrastructures, IoT/cloud infrastructures and related services can be classified to the following models:

1. **Infrastructure-as-a-Service (IaaS) IoT/Clouds**: These services provide the means for accessing sensors and actuator in the cloud. The associated business model involves the IoT/Cloud provide to act either as data or sensor provider. IaaS services for IoT provide access control to resources as a prerequisite for the offering of related pay-as-you-go services.
2. **Platform-as-a-Service (PaaS) IoT/Clouds**: This is the most widespread model for IoT/cloud services, given that it is the model provided by all public IoT/cloud infrastructures outlined above. As already illustrate most public IoT clouds come with a range of tools and related environments for applications development and deployment in a cloud environment. A main characteristic of PaaS IoT services is that they provide access to data, not to hardware. This is a clear differentiator comparing to IaaS.

3. **Software-as-a-Service (SaaS) IoT/Clouds**: SaaS IoT services are the ones enabling their uses to access complete IoT-based software applications through the cloud, on-demand and in a pay-as-you-go fashion. As soon as sensors and IoT devices are not visible, SaaS IoT applications resemble very much conventional cloud-based SaaS applications. There are however cases where the IoT dimension is strong and evident, such as applications involving selection of sensors and combination of data from the selected sensors in an integrated applications. Several of these applications are commonly called Sensing-as-a- Service, given that they provide on-demand access to the services of multiple sensors. Note that SaaS IoT applications are typically built over a PaaS infrastructure and enable utility based business models involving IoT software and services.

These definitions and examples provide an overview of IoT and cloud convergence and why it is important and useful. More and more IoT applications are nowadays integrated with the cloud in order to benefit from its performance, business agility and pay-as-you-go characteristics. In following chapters of the tutorial, we will present how to maximize the benefits of the cloud for IoT, through ensuring semantic interoperability of IoT data and services in the cloud, thus enabling advanced data analytics applications, but also integration of a wide range of vertical (silo) IoT applications that are nowadays available in areas such as smart energy, smart transport

and smart cities. We will also illustrate the benefits of IoT/cloud integration for specific areas and segments of IoT, such as IoT-based wearablecomputing.

WAMP for IoT

Web Application Messaging Protocol (WAMP) is a sub-protocol of Websocket which provides publish-subscribe and remote procedure call (RPC) messaging patterns.

WAMP

1. **Transport:** Transport is channel that connects two peers.

2. **Session:** Session is a conversation between two peers that runs over atransport.

3. **Client:** Clients are peers that can have one or more roles. In publish-subscribe model client can have followingroles:

a) Publisher: Publisher publishes events (including payload) to the topic maintained by thebroker.

b) Subscriber: Subscriber subscribes to the topics and receives the events including the payload.

In RPC model client can have following roles: –

1. **Caller:** Caller issues calls to the remote procedures along with call arguments. – Callee: Callee executes the procedures to which the calls are issued by the caller and returns the results back to the caller. • Router: Routers are peers that perform generic call and event routing. In publish-subscribe model Router has the role of a Broker: – Broker: Broker acts as a router and routes messages published to a topic to all subscribers subscribed to thetopic.

In RPC model Router has the role of a Broker: –

1. Dealer: Dealer acts a router and routes RPC calls from the Caller to the Callee and routes results from Callee toCaller.

2. Application Code: Application code runs on the Clients (Publisher, Subscriber, Callee or Caller).

Amazon EC2 – Python Example

Boto is a Python package that provides interfaces to Amazon Web Services (AWS). In this example, a connection to EC2 service is fi rst established by calling boto.ec2.connect_to_region. The EC2 region, AWS access key and AWS secret key are passed to this function. After connecting to EC2 , a new instance is launched using the conn.run_instances function. The AMI-ID, instance type, EC2 key handle and security group are passed to this function

```
#Python program for launching an EC2 instance
import boto.ec2
from time import sleep
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"
AMI_ID = "ami-d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

conn = boto.ec2.connect_to_region(REGION, aws_access_key_id=ACCESS_KEY,
                                  aws_secret_access_key=SECRET_KEY)

reservation = conn.run_instances(image_id=AMI_ID, key_name=EC2_KEY_HANDLE,
                                 instance_type=INSTANCE_TYPE,
                                 security_groups = [ SECGROUP_HANDLE, ] )
```

Amazon AutoScaling – Python Example

1. **AutoScaling Service:** A connection to AutoScaling service is first established by calling boto.ec2.autoscale.connect_to_regionfunction.

2. **Launch Configuration**: After connecting to AutoScaling service, a new launch configuration is created by calling conn.create_launch_con f iguration. Launch configuration contains instructions on how to launch new instances including the AMI- ID, instance type, security groups,etc.

3. **AutoScaling Group** : After creating a launch configuration, it is then associated with a new AutoScaling group. AutoScaling group is created by calling conn.create_auto_scaling_group. The settings for AutoScaling group such as the maximum and minimum number of instances in the group, the launch configuration, availability zones, optional load balancer to use with the group,etc.

Amazon AutoScaling – Python Example

#Creating auto-scaling policies
scale_up_policy = ScalingPolicy(name='scale_up', adjustment_type='ChangeInCapacity', as_name='My-Group', scaling_adjustment=1,
cooldown=180)
scale_down_policy =ScalingPolicy(name='scale_down', adjustment_type='ChangeInCapacity', as_name='My-Group', scaling_adjustment=-1, cooldown=180)
conn.create_scaling_policy(scale_up_policy) conn.create_scaling_policy(scale_down_policy)

AutoScaling Policies:

1. After creating an AutoScaling group, the policies for scaling up and scaling down are defined.

2. In this example, a scale up policy with adjustment type ChangeInCapacity and scaling_ad justment = 1 isdefined.

3. Similarly a scale down policy with adjustment type ChangeInCapacity and scaling_ad justment = -1 isdefined.

CloudWatch Alarms

#Connecting to CloudWatch cloudwatch = boto.ec2.cloudwatch.connect_to_region(REGION, aws_access_key_id=ACCESS_KEY,aws_secret_access_key=SECRET_KEY)
alarm_dimensions = {"AutoScalingGroupName": 'My-Group'}
#Creating scale-up alarm scale_up_alarm = MetricAlarm(name='scale_up_on_cpu', namespace='AWS/EC2', metric='CPUUtilization', statistic='Average', comparison='>', threshold='70', period='60', evaluation_periods=2, alarm_actions=[scale_up_policy.policy_arn], dimensions=alarm_dimensions) cloud watch.create_alarm(scale_up_alarm)
#Creating scale-down alarm scale_down_alarm =MetricAlarm (name='scale_down_on_cpu',namespace='AWS/EC2', metric='CPUUtilization', statistic='Average', comparison='<',threshold='40', period='60', evaluation_periods=2, alarm_actions=[scale_down_policy.policy_arn], dimensions=alarm_dimensions) cloud watch.create_alarm(scale_down_alarm)

1. With the scaling policies defined, the next step is to create Amazon Cloud Watch alarms that trigger these policies.

2. The scale up alarm is defined using the CPU Utilization metric with the Average statistic and threshold greater 70% for a period of 60 sec. The scale up policy created previously is associated with this alarm. This alarm is triggered when the average CPU utilization of the instances in the group becomes greater than 70% for more than 60seconds.

3. The scale down alarm is defined in a similar manner with a threshold less than50%.

Python for Map Reduce

#Inverted Index Mapper in Python

#!/usr/bin/env python import sys for line in sys.stdin: doc_id, content = line.split(‘‘) words = content.split() for word in words: print ‘%s%s‘ % (word, doc_id)

The example shows inverted index mapper program. The map function reads the data from the standard input (stdin) and splits the tab-limited data into document- ID and contents of the document. The map function emits key-value pairs where key is each word in the document and value is the document-ID.

Python for MapReduce

#Inverted Index Reducer in Python

#!/usr/bin/env python import sys current_word = None current_docids = [] word =
None

for line in sys.stdin: # remove leading and trailing whitespace line = line.strip() # parse the input we got from mapper.py word, doc_id = line.split(‘‘) if current_word
== word: current_docids.append(doc_id) else: if current_word: print ‘%s%s‘ % (current_word, current_docids) current_docids = [] current_docids.append(doc_id) current_word = word

The example shows inverted index reducer program. The key-value pairs emitted by the map phase are shuffled to the reducers and grouped by the key. The reducer reads the key-value pairs grouped by the same key from the standard input (stdin) and creates a list of document-IDs in which the word occurs. The output of reducer contains key value pairs where key is a unique word and value is the list of document-IDs in which the word occurs.

Python Packages of Interest

1. **JSON:** JavaScript Object Notation (JSON) is an easy to read and write data- interchange format. JSON is used as an alternative to XML and is is easy for machines to parse and generate. JSON is built on two structures - a collection of name-value pairs (e.g. a Python dictionary) and ordered lists of values (e.g.. a Pythonlist).

2. **XML:** XML (Extensible Markup Language) is a data format for structured document interchange. The Python minidom library provides a minimal implementation of the Document Object Model interface and has an API similar to that in otherlanguages.

3. **HTTPLib & URLLib:** HTTPLib2 and URLLib2 are Python libraries used in network/internetprogramming

4. **SMTPLib:** Simple Mail Transfer Protocol (SMTP) is a protocol which handles sending email and routing e-mail between mail servers. The Python smtplib module provides an SMTP client session object that can be used to send email.

5. **NumPy:**NumPy is a package for scientific computing in Python. NumPy provides support for large multi-dimensional arrays andmatrices

6. **Scikit-learn:** Scikit-learn is an open source machine learning library for Python that provides implementations of various machine learning algorithms for classification, clustering, regression and dimension reduction problems.

Python Web Application Framework - Django

Django is an open source web application framework for developing web applications in Python. A web application framework in general is a collection of solutions, packages and best practices that allows development of web applications and dynamic websites. Django is based on the Model-Template-View architecture and provides a separation of the data model from the business rules and the user interface. Django provides a unified API to a database backend. Thus web applications built with Django can work with different databases without requiring any code changes. With this flexibility in web application design combined with the powerful capabilities of the Python language and the Python ecosystem, Django is best suited for cloud applications. Django consists of an object-relational mapper, a web templating system and a regular-expression based URL dispatcher.

Django Architecture

Django is Model-Template-View (MTV) framework.

1. Model: The model acts as a definition of some stored data and handles the interactions with the database. In a web application, the data can be stored in a relational database, non-relational database, an XML file, etc. A Django model is a Python class that outlines the variables and methods for a particular type of data.

2. Template: In a typical Django web application, the template is simply an HTML page with a few extra placeholders. Django‘s template language can be used to create various forms of text files (XML, email, CSS, Java script, CSV,etc.)

3. View :The view ties the model to the template. The view is where you write the code that actually generates the web pages. View determines what data is to be displayed, retrieves the data from the database and passes the data to the template.

Case studies illustrating IoT design Case Study in IoT: Home Automation

An IoT software-based approach on the field of Home Automation. Common use-cases include measuring home conditions, controlling home appliances and controlling home access through RFID cards as an example and windows through servo locks. However, the main focus of this paper is to maximize the security of homes through IoT. More specifically, monitoring and controlling servo door locks, door sensors, surveillance cameras, surveillance car and smoke detectors, which help ensuring and maximizing safety and security of homes.

A user has the following features through a mobile application in which he/she:

1. can turn on or o_ LED lights and monitor the state of the LED.

2. can lock and unlock doors through servo motors and monitor if the doors are locked or unlocked.

3. can monitor if the doors are closed or opened through IR sensors.

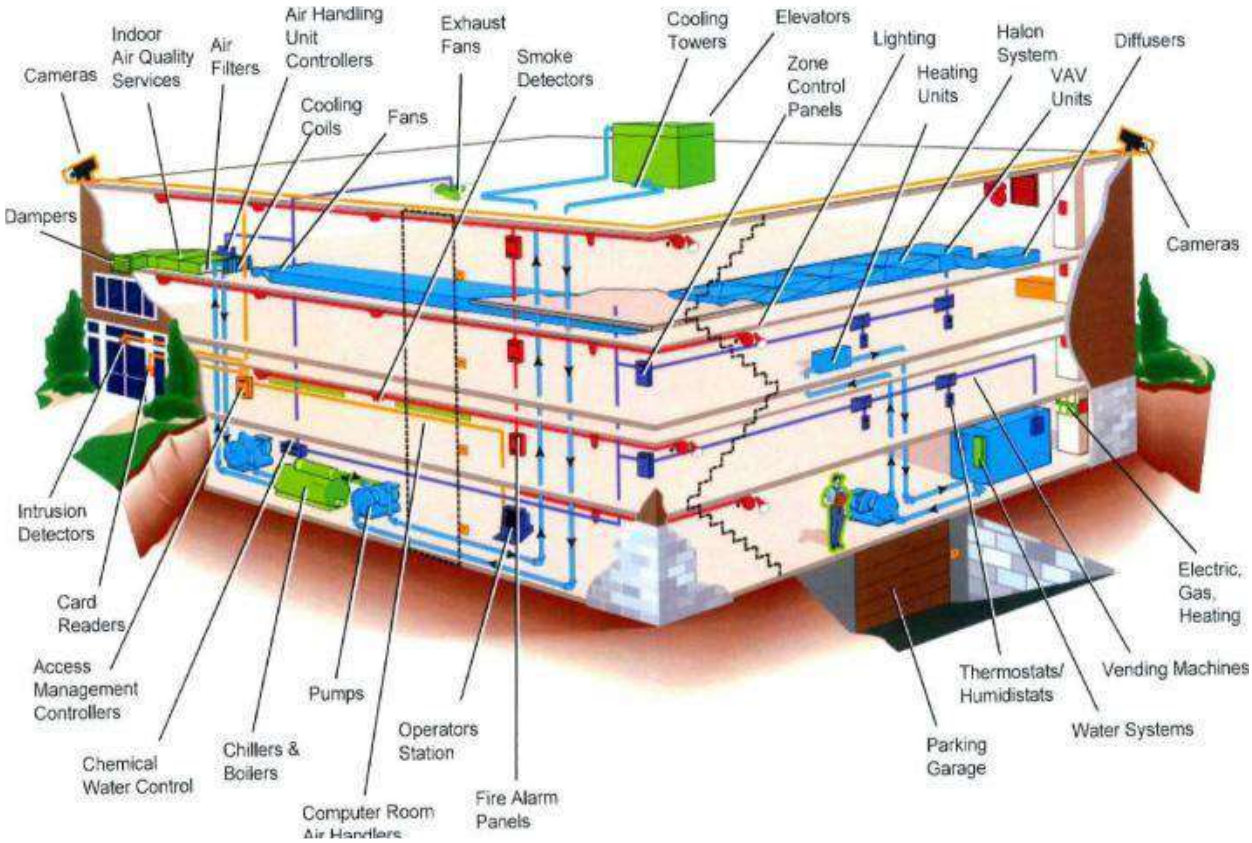
4. is notified through email if the door is left open for too long.

5. is notified of who entered through the door as the camera captures the face image and send it to him/her via email.

6. is notified through email if the _re detector detects smoke.

7. is able to control the surveillance car from anywhere to monitor his/her home.

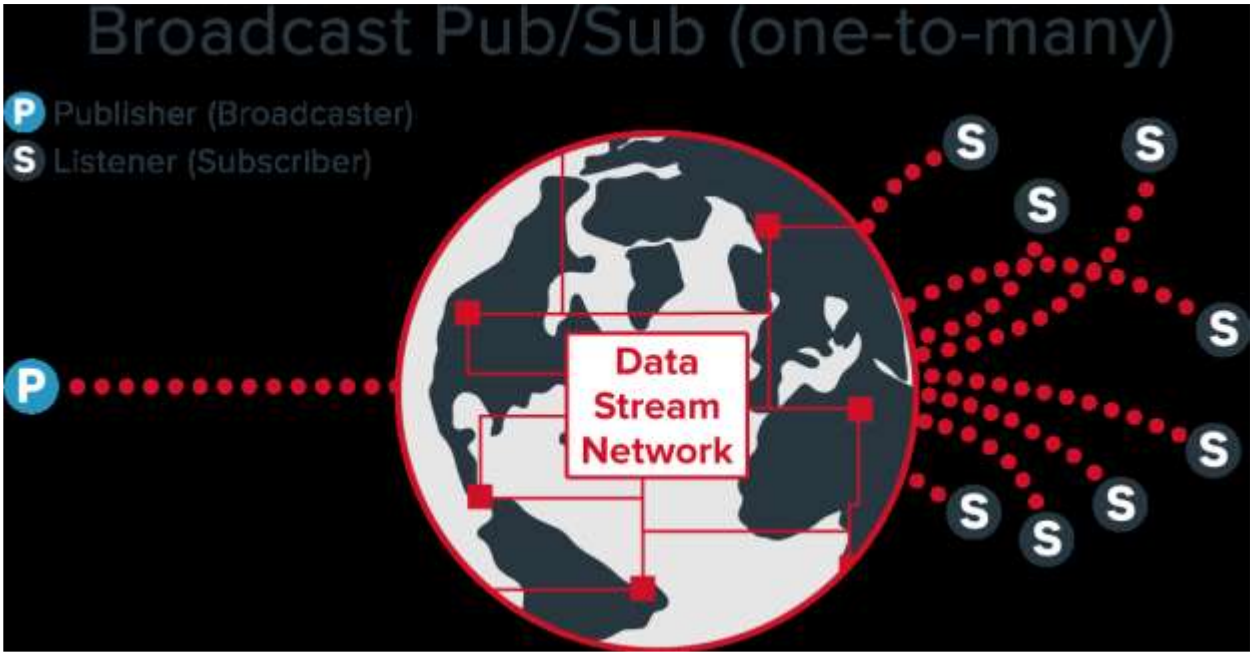
As the field of Home Automation through IoT is a wide application in a very wide and challenging field due to the reasons mentioned in the previous paragraphs, I chose to work on that field as part of this thesis, specifically in maintaining and ensuring security and safety inside home.



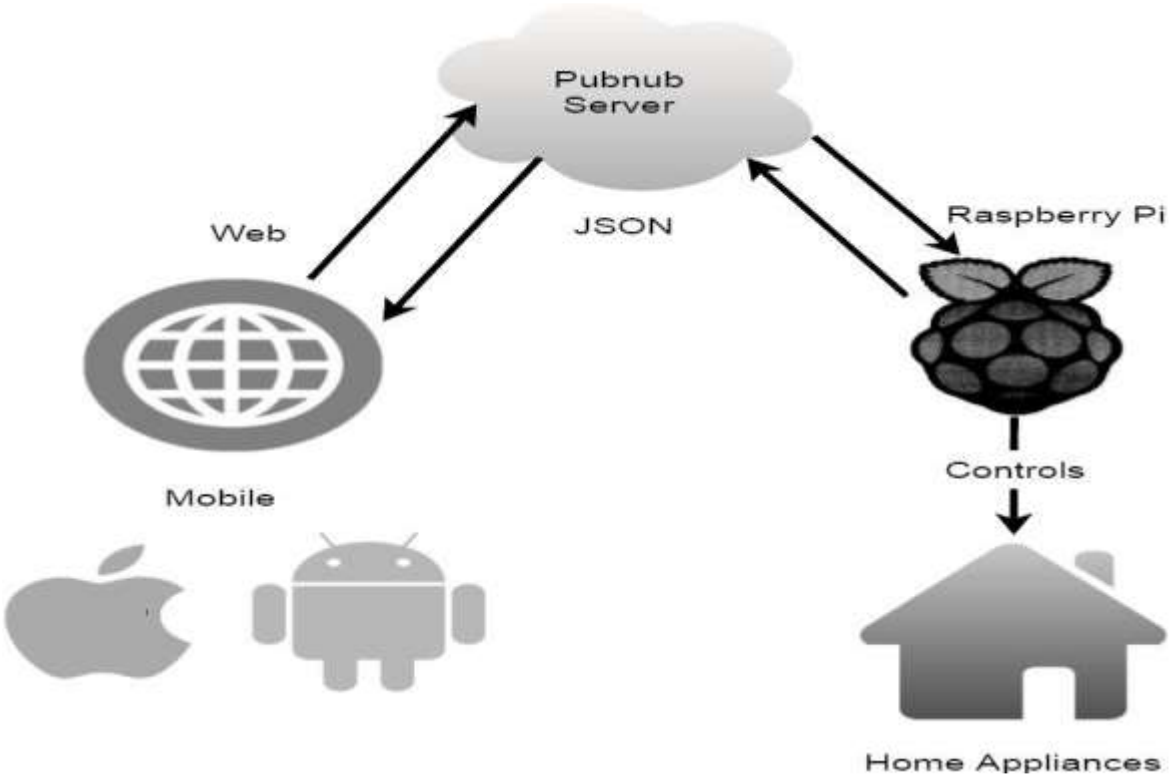
IoT aims in creating a network between objects embedded with sensors, that can store, analyze, communicate and exchange data together over the internet. This leads to efficient industry, manufacturing, efficient energy management, resource management, accurate health

care, smarter business decisions based on analyzed data, safer driving through smart cars that are able to communicate together, smart home automation and countless more applications.

The system designed for the home automation project presented in this paper needs a control unit, a computer, to be able to control the different electrical devices connected to it. Raspberry Pi, is a credit-card tiny computer, that can be plugged to a monitor, uses standard keyboard and mouse, that enables people of different ages learn how to program.



Illustrates the publish/subscribe model provided by PubNub



Illustrates the system architecture used in this home automation project.

To simplify the publish/subscribe model along with the system architecture used in this Home Automation project, here is the explanation of the steps of constructing it: Different sensors, cameras and servo motors were connected to the Raspberry Pi. It was programmed to collect and publish the data, in the form of JSON string, acquired from these devices to PubNub. Data is published from the Raspberry Pi by providing it with the "publish key" and the "channel name". The data is sent to the channel provided by PubNub servers, and forwarded by PubNub to the subscribers of this channel.

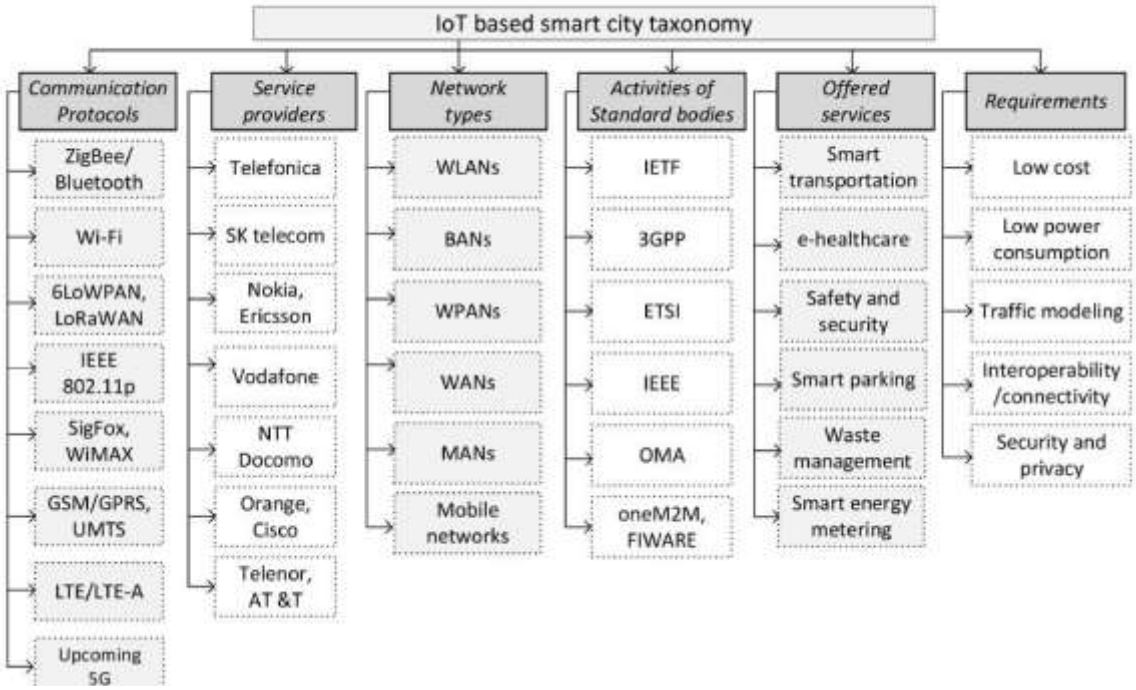
The subscriber in this scenario, of a user acquiring data and readings by the sensors and monitoring devices, is the web/mobile application. The "subscription key" and "channel name" is embedded in the web/mobile application's code. Allowing it to receive messages forwarded by PubNub. On the other hand, in a scenario where the user wants to send a command to home appliances, controlling the LED lights for example, the web/mobile application is the publisher provided by the "publish key" and the "channel name". The commend is sent in the form of JSON string to PubNub servers, while the "subscription key" and "channel name" is embedded in the Raspberry Pi code. This allows the Raspberry Pi to receive any published strings on the channel it is subscribed to. Upon receiving the JSON string, the Raspberry Pi take the action

specified by that string. This allows full control and monitoring of all devices connected to the Raspberry Pi by the user.

Case Study in IoT: Smart Cities

The Internet-of-Things (IoT) is the novel cutting-edge technology which proffers to connect plethora of digital devices endowed with several sensing, actuation and computing capabilities with the Internet, thus offers manifold new services in the context of a smart city. The appealing IoT services and big data analytics are enabling smart city initiatives all over the world. These services are transforming cities by improving infrastructure, transportation systems, reduced traffic congestion, waste management and the quality of human life. In this paper, we devise a taxonomy to best bring forth a generic overview of IoT paradigm for smart cities, integrated information and communication technologies (ICT), network types, possible opportunities and major requirements. Moreover, an overview of the up-to-date efforts from standard bodies is presented. Later, we give an overview of existing open source IoT platforms for realizing smart city applications followed by several exemplary case studies. In addition, we summarize the latest synergies and initiatives worldwide taken to promote IoT in the context of smart cities. Finally, we highlight several challenges in order to give future research directions

An illustration of IoT based smart city



A representation of IoT based smart city taxonomy

IOT BASED SMART CITY TAXONOMY

This section presents a taxonomy of IoT based smart cities which categorizes the literature on the basis of existing communication protocols, major service providers, network types, standardization efforts, offered services, and crucial requirements.

Communication Protocols

IoT based smart city realization significantly relies on numerous short and wide range communication protocols to transport data between devices and backend servers. Most prominent short range wireless technologies include Zig-Bee, Bluetooth, Wi-Fi, Wireless Metropolitan Area Network (WiMAX) and IEEE 802.11p which are primarily used in smart metering, e-healthcare and vehicular communication. Wide range technologies such as Global System for Mobile communication (GSM) and GPRS, Long-Term Evolution (LTE), LTE-Advanced are commonly utilized in ITS such as vehicle-to infrastructure (V2I), mobile e-healthcare, smart grid and infotainment services. Additionally, LTE-M is considered as an evolution for cellular IoT (C-IoT). In Release 13, 3GPP plans to further improve coverage, battery lifetime as well as device complexity [7]. Besides well-known existing protocols, LoRa alliance standardizes the LoRaWAN protocol to support smart city applications to primarily ensure interoperability between several operators. Moreover, SIGFOX is an ultra narrowband radio technology with full star-based infrastructure offers a high scalable global network for

realizing smart city applications with extremely low power consumption. A comparative summary2 of the major communicationprotocols.

Service Providers

Pike Research on smart cities estimated this market will grow to hundreds of billion dollars by 2020, with an annual growth of nearly 16 billion. IoT is recognized as a potential source to increase revenue of service providers. Thus, well-known worldwide service providers have already started exploring this novel cutting edge communication paradigm. Major service providers include Telefonica, SK telecom, Nokia, Ericsson, Vodafone, NTT Docomo, Orange, Telenor group and AT&T which offer variety of services and platforms for smart city applications such as ITS and logistics, smart metering, home automation and e-healthcare.

Network Types

IoT based smart city applications rely on numerous network topologies to accomplish a fully autonomous environment. The capillary IoT networks offer services over a short range. Examples include wireless local area networks (WLANs), BANs and wireless personal area networks (WPANs). The application areas include indoor e-healthcare services, home automation, street lighting. On the other hand, applications such as ITS, mobile e-healthcare and waste management use wide area networks (WANs), metropolitan area networks (MANs),and mobile communication networks. The above networks pose distinct features in terms of data, size, coverage, latency requirements, and capacity.

Case Study in IoT: Smart Environment

The rapid advancements in communication technologies and the explosive growth of Internet of Things (IoT) have enabled the physical world to invisibly interweave with actuators, sensors, and other computational elements while maintaining continuous network connectivity. The continuously connected physical world with computational elements forms a smart environment. A smart environment aims to support and enhance the abilities of its dwellers in executing their tasks, such as navigating through unfamiliar space and moving heavy objects for the elderly, to name a few. Researchers have conducted a number of efforts to use IoT to facilitate our lives and to investigate the effect of IoTbased smart environments on human life. This paper surveys the state-of-the-art research efforts to enable the IoT-based smart environments. We categorize and classify the literature by devising a taxonomy based on communication enablers, network types, technologies, local area wireless standards, objectives, and characteristics. Moreover, the paper highlights the unprecedented opportunities brought about by IoT-based smart environments and their effect on human life. Some reported case studies from different enterprises are also presented. Finally, we discuss open research challenges for enabling IoT-based smart environments. Immense developments and increasing miniaturization of computer technology have enabled tiny sensors and processors to be integrated into everyday objects. This advancement is further supported by tremendous developments in areas such as portable appliances and devices, pervasive computing, wireless

sensor networking, wireless mobile communications, machine learning-based decision making, IPv6 support, human computer interfaces, and agent technologies to make the dream of smart environment a reality. A smart environment is a connected small world where sensor-enabled connected devices work collaboratively to make the lives of dwellers comfortable. The term smart refers to the ability to autonomouslyobtain and applies knowledge; and the term environment refers to the surroundings. Therefore, a smart environment is one that is capable of obtaining knowledge and applying it to adapt according to its inhabitants‘ needs to ameliorate their experience of that environment.

The functional capabilities of smart objects are further enhanced by interconnecting them with other objects using different wireless technologies. In this context, IPv6 plays a vital role because of several features, including better security mechanisms, scalability in case of billion of connected devices, and the elimination of NAT barriers1. This concept of connecting smart objects with the Internet was first coined by Kevin Ashton as ―Internet of Things‖ (IoT).

Nowadays, IoT is receiving attention in a number of fields such as healthcare, transport, and industry, among others. Several research efforts have been conducted to integrate IoT with smart environments. The integration of IoT with a smart environment extends the capabilities of smart objects by enabling the user to monitor the environment from remote sites. IoT can be integrated with different smart environments based on the application requirements. The work on IoT-based smart environments can generally be classified into the following areas: a) smart cities, b) smart homes, c) smart grid, d) smart buildings, e) smart transportation, f) smart health, and g) smart industry. illustrates the IoT-based smart environments. The taxonomy of the IoT based smart environment. The devised taxonomy is based on the following parameters: communication enablers, network types, technologies, wireless standards, objectives, and characteristics

Communication Enablers

Communication enablers refer to wireless technologies used to communicate across the Internet. The key wireless Internet technologies are WiFi, 3G, 4G, and satellite. WiFi is mainly used in smart homes, smart cities, smart transportation, smart industries, and smart building environments; whereas, 3G and 4G are mainly used in smart cities and smart grid environments. Satellites are used in smart transportation, smart cities, and smart grid environments. Table presents the comparative summary of the communication technologies used in IoT based smart environments.

Network Types

IoT-based smart environments rely on different types of networks to perform the collaborative tasks for making the lives of inhabitants more comfortable. The main networks are wireless local area networks (WLANs), wireless personal area networks (WPANs), wide area networks (WANs), metropolitan area networks (MANs), and wireless regional area networks (WRANs). These networks have different characteristics in terms of size, data transfer, and supported reach ability.

Technologies

IoT-based smart environments leverage various technologies to form a comfortable and suitable ecosystem. These technologies are include sensing, communication, data fusion, emerging computing, and information security. Sensing technologies are commonly used to acquire data from various locations and transmit it using communication technologies to a central location. The emerging computing technologies, such as cloud computing and fog computing, deployed in the central location, leverage the data fusion technologies for integrating the data coming from heterogeneous resources. In addition, smart environments also use information security technologies to ensure data integrity and user privacy.

Local Area Wireless Standards

The commonly used local area wireless standards in IoT-based smart environments are IEEE 802.11, IEEE 802.15.1, and IEEE 802.15.4. These standard technologies are used inside the smart environment to transfer the collected data among different devices. IEEE 802.11 is used in smart homes, smart buildings, and smart cities. IEEE 802.15.1 and IEEE 802.15.4 have relatively shorter coverage than IEEE 802.11 and are used mainly in sensors and other objects deployed in the smart environments.