MODULE-2

IOT and M2M: Introduction: M2M, Difference between IoT and M2M, SDN and NFV for IOT, IOT System Management with NETCONF-YANG, Need for IOT Systems Management, Simple Network Management Protocol (SNMP), Network operator requirements, NETCONF, YANG, IoT Systems Management with NETCONF-YANG.

IOT and M2M

2.1 : Introduction

In module 1 learned about the definition and characteristics of Internet of Things (IoT). Another term which is often used synonymously with IoT is Machine-to-Machine (M2M). Though IoT and M2M are often used interchangeably, these terms have evolved from different backgrounds. This chapter describes some of the differences and similarities between IoT and M2M.

2.2 : M2M

Machine-to-Machine (M2M) refers to networking of machines (or devices) for the purpose of remote monitoring and control and data exchange. Figure 3.1 shows the end-to-end architecture for M2M systems comprising of M2M area networks, communication network and application domain.

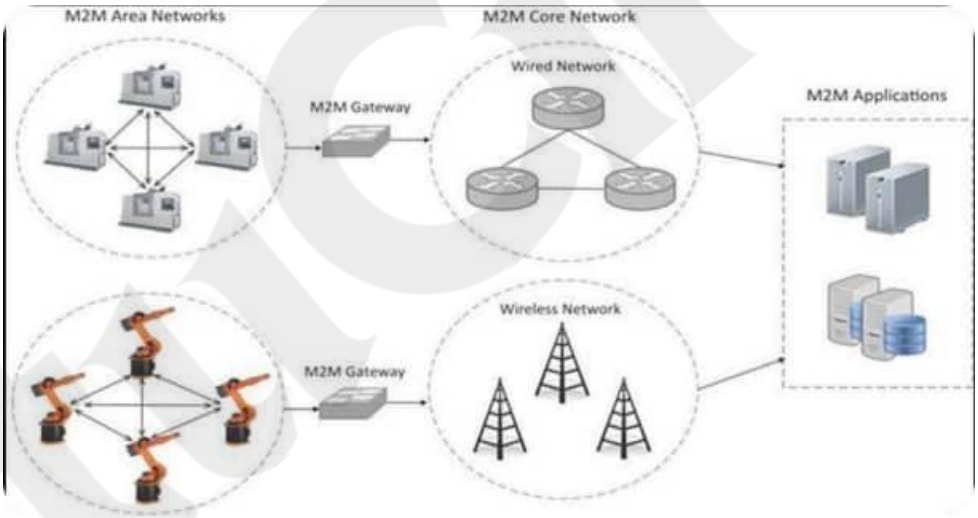


Figure 3.1: M2M system architecture

Machine-to-Machine (M2M) communication means machines or devices are connected to share data and control each other without human involvement.
These systems are made up of three main parts:

1. M2M area network (local machines)
2. Communication network (connects different areas)
3. Application (where data is used)

In the M2M area network, devices (called M2M nodes) have built-in sensors, actuators, and communication modules. They use short-range communication protocols like ZigBee, Bluetooth, Modbus, and others to talk to each other locally. However, these protocols are usually non-IP based,

while the external communication network (which connects different M2M networks) uses IP-based systems like the internet. This creates a problem, as non-IP devices can't directly connect with IP networks. To solve this, an M2M gateway is used. The gateway acts as a bridge, converting non-IP communication into IP format, so that devices in different locations can communicate with each other through the internet.

Figure 3.2 shows how an M2M gateway works. M2M nodes (devices) talk to the gateway using their own local communication methods. The gateway's job is to **convert these local messages into internet-friendly messages (IP-based)** so the devices can connect to the outside world. In this way, the gateway acts like a **translator** between the local devices and the internet. Because of the gateway, each device in the local network can be seen and used by devices in other networks, as if they were directly connected.

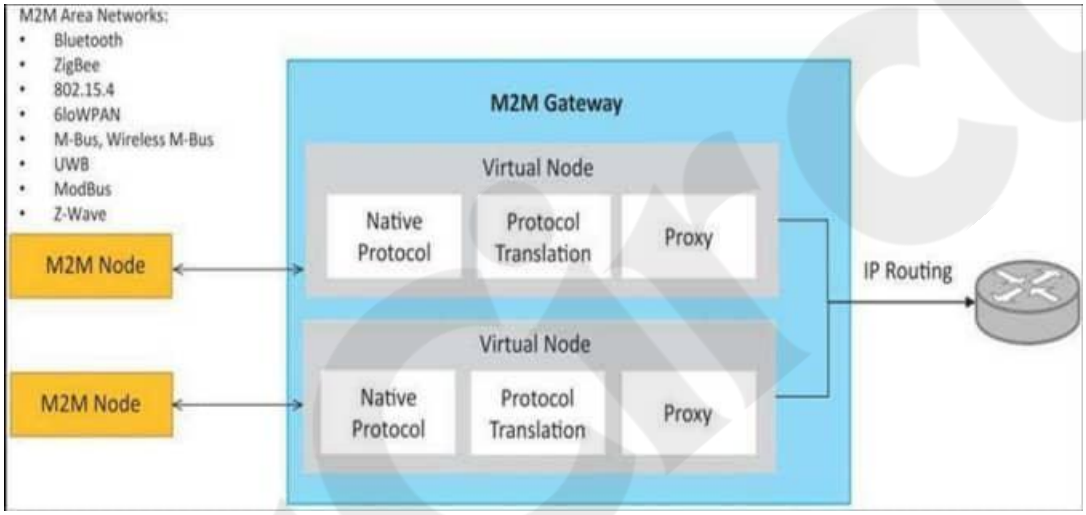


Figure 2.2: block diagram of an M2M gateway.

M2M data is used in apps like remote monitoring, service management, and business tools. It is helpful in areas like smart homes, factories, and smart electricity systems. Each area needs its own way to collect and use the data.

2.3 Differences between IoT and M2M

1. Communication Protocols

- M2M uses **proprietary or non-IP** protocols for communication within local networks.
- Common M2M protocols: **ZigBee, Bluetooth, ModBus, M-Bus, PLC, 6LoWPAN, IEEE 802.15.4, Z-Wave**.
- M2M focuses on protocols **below the network layer**.
- IoT uses **IP-based protocols** such as **HTTP, CoAP, WebSockets, MQTT, XMPP, DDS, AMQP**.
- IoT focuses on protocols **above the network layer as shown in the figure 3.3**.

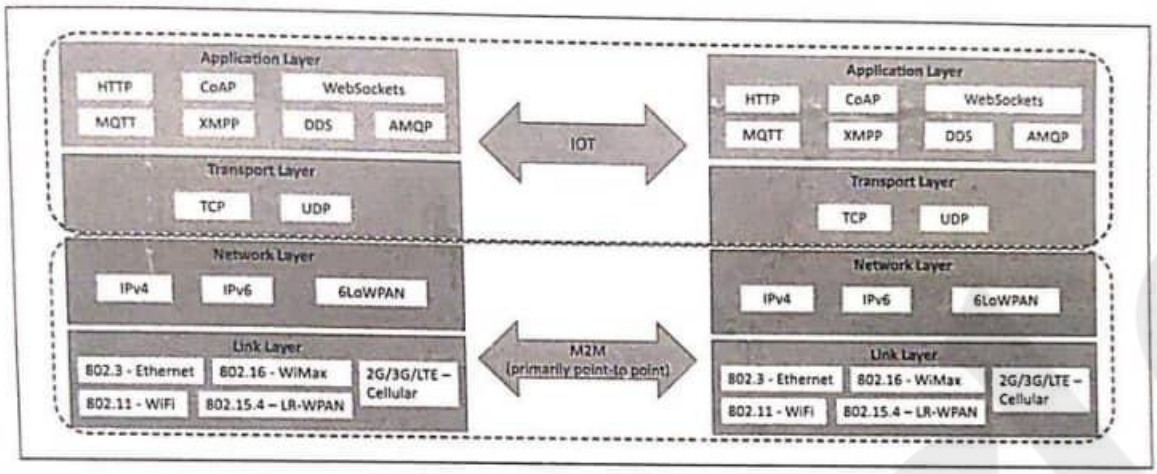


Figure 3.3: Communication in IoT is IP-based whereas M2M uses non-IP based networks. Communication within M2M area networks is based on protocols below the network layer whereas IoT is based on protocols above the network layer.

2. Machines in M2M vs Things in IoT

- In **IoT**, "Things" are physical objects with **unique identifiers** (IP or MAC addresses).
- These things can **sense**, **process**, and **communicate** with their environment.
- IoT includes **various devices** (e.g., fire alarms, door sensors, lighting controls).
- IoT systems are **heterogeneous** (different types of devices).
- **M2M systems** have **homogeneous** devices (same type) within one network.

3. Hardware vs Software Emphasis

- **M2M** focuses more on **hardware** with embedded modules.
- **IoT** focuses more on **software** for:
  - Sensor data collection
  - Data analysis
  - Communication with the **cloud** via IP.

4. Data Collection & Analysis

- **M2M data** is stored **locally** (on-premises).
- **IoT data** is stored in the **cloud** (public, private, or hybrid).
- IoT includes **analytics components** that:
  - Analyze data
  - Store results in the cloud
  - Visualize results through cloud apps
- **Centralized controller** in IoT monitors node status and sends commands.
- **Observer nodes** can process data but do **not** control devices.

5. Applications

- **M2M** supports on-premises apps like:
  - Diagnosis systems
  - Service management
  - Enterprise solutions
- **IoT** supports cloud apps like:
  - Analytics

- Enterprise applications
- Remote monitoring and management

- IoT uses **real-time and batch data analysis** due to **large-scale data collection**.

2.4 SDN and NFV for IoT

In this you will learn about software define networking (SDN) and Network Function Virtualization (NFV) and their applications for IoT.

2.4.1 Software Define Networking (SDN)

It is a new way of designing computer networks.

- In SDN, the network is divided into two parts:
  1. Control Plane – Decides how data should move.
  2. Data Plane – Actually moves the data.
- In old or conventional networks, both parts are inside the same hardware device (like switches or routers). These devices are becoming more complex because they use many different protocols and special hardware Figure 3.6 shows the conventional network architecture built with specialized hardware (switches, routers, etc.).
- In SDN, the control plane is separated from the data plane. The control part is centralized — meaning one main controller handles decisions for the whole network.
- This setup makes networks easier to manage, update, and change.

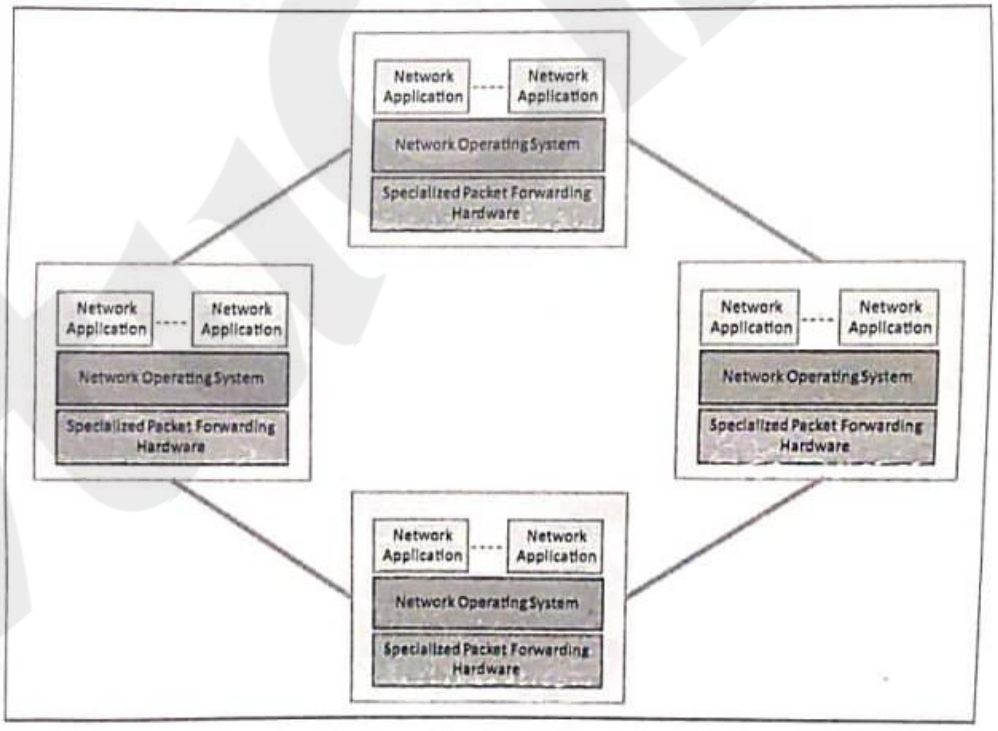


Figure 3.6: Conventional network architecture

The limitations of the conventional network architectures are as below

- Complex network devices
- Management overhead
- Limited Scalability

SDN attempts to create network architectures that are simpler, inexpensive, scalable, agile and easy to manage. Figures 3.7 and 3.8 show the SDN architecture and the SDN layers in which the control and data planes are decoupled and the network controller is centralized. Software-based SDN controllers maintain a unified view of the network and make configuration, management and provisioning simpler. The underlying infrastructure in SDN uses simple packet forwarding hardware as opposed to specialized hardware in conventional networks. The underlying network infrastructure is abstracted from the applications. Network devices become simple with SDN as they do not require implementations of a large number of protocols. Network devices receive instructions from the SDN controller on how to forward the packets. These devices can be simpler and cost less as they can be built from standard hardware and software components.

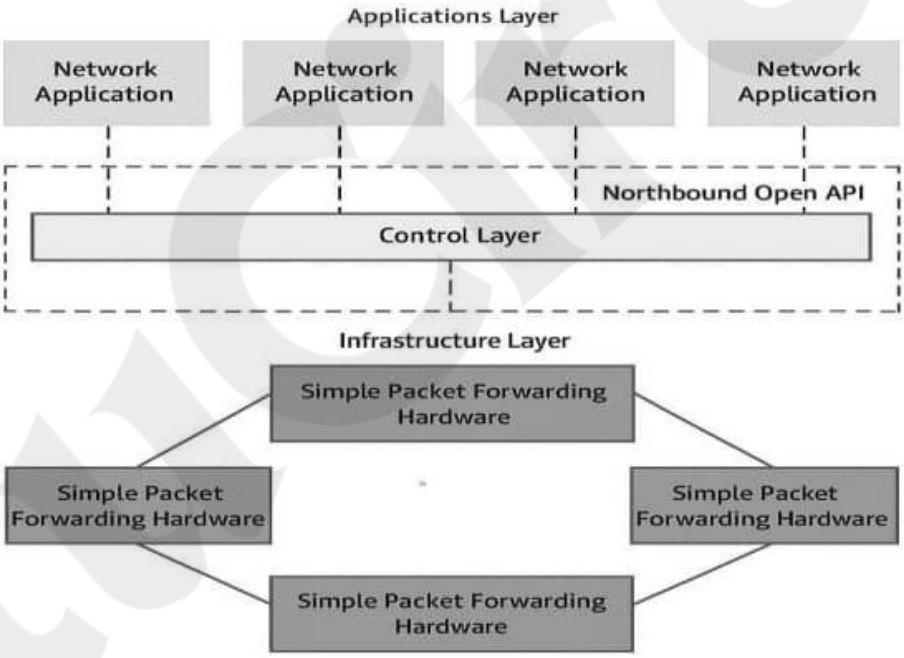


Figure 3.7: SDN layers

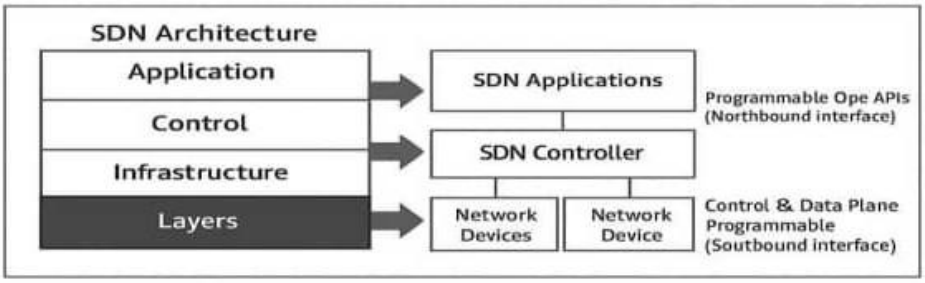


Figure 3.8: SDN layers

Key elements of SDN:

1. **Centralized Network Controller:**
   In SDN, the control part of the network is separated and placed in one central location. This makes it easy for network admins to manage and change the network quickly.
   They don't need to wait for hardware companies to add new features — they can add new functions using software through open tools (APIs). This helps make the network faster and more innovative.
2. **Programmable Open APIs:**
   SDN provides **open tools (called APIs)** that allow communication between:
   - Applications (that control or monitor the network)
   - The control system (the SDN controller)
     These tools let you build network features like **routing**, **security**, and more easily.
3. **Standard Communication Interface (OpenFlow):**
   OpenFlow is a protocol used in SDN (Software-Defined Networking) that lets the **controller** talk to the **network devices** (like switches).
   It connects the **control layer** (brains) with the **infrastructure layer** (hardware that moves data). This connection is called the **Southbound interface**.
   How OpenFlow Works:
   - OpenFlow allows the controller to directly control how network devices handle data.
   - It uses **"flows"** to manage network traffic. A **flow** is a rule that says: *"If traffic looks like this, do this."*
   - These flows can be:
     - **Static** (fixed) or
     - **Dynamic** (changeable by software)

OpenFlow Components:

- A switch using OpenFlow has:
  - One or more **flow tables**: where rules are stored
  - A **group table**: for handling multiple flows
  - A **channel** to communicate with the controller

  Figure 3.9 shows the components of an OpenFlow switch comprising of one or more flow tables and a group table, which perform packet lookups and forwarding, and OpenFlow channel to an external controller.

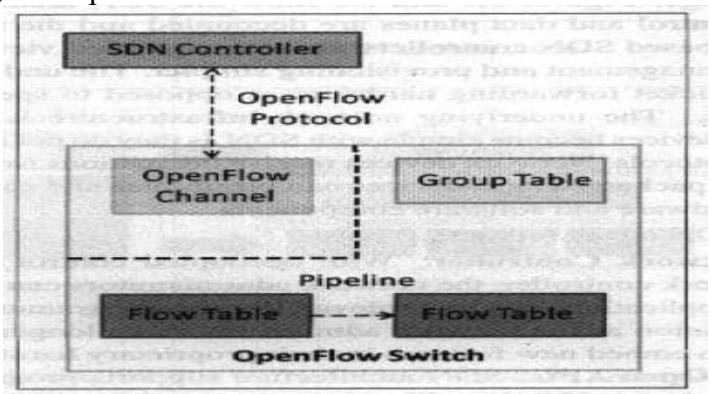


Figure 3.9: OpenFlow switch

What the Controller Can Do:

- The SDN controller can:
  - **Add**
  - **Update**
  - **Delete** flow rules in the switch's flow table

Flow Table:

- A **flow table** has a list of flow entries.
- Each entry contains:
  - **Match fields**: what kind of traffic to look for
  - **Counters**: for tracking
  - **Actions**: what to do with matching traffic

Figure 3.10 shows an example of an OpenFlow flow table.

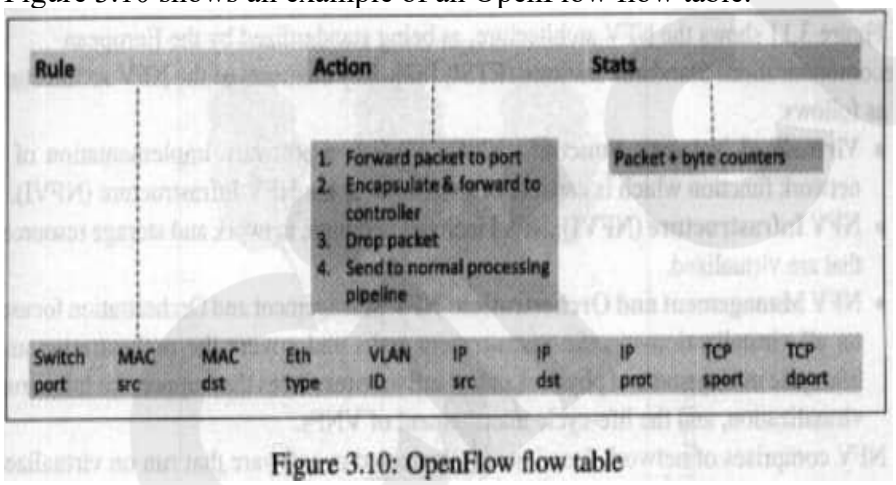


Figure 3.10: OpenFlow flow table

- The device checks the flow table from top to bottom and can move through multiple tables to decide how to handle packets.

2.4.2 Network Function Virtualization (NFV)

NFV is a technology that uses virtualization to replace physical network devices (like firewalls, routers, etc.) with software running on regular servers, switches, and storage. It helps combine different types of network devices into one standard, high-capacity system.

Relation with SDN (Software-Defined Networking):

- **NFV and SDN can work together** and support each other:
  - **NFV gives the platform** (infrastructure) for SDN to run.
  - **SDN controls the network** intelligently.
- But they are **not dependent** on each other:
  - NFV can work **without SDN**
  - SDN can work **without NFV**

Figure 3.11 shows the NFV architecture, as being standardized by the European Telecommunications Standards Institute (ETSI).

Key elements of the NFV architecture are as follows:

- **Virtualized Network Function (VNF):** VNF is a software implementation of a network function which is capable of running over the NFV Infrastructure (NFVI).

- **NFV Infrastructure (NFVI):** NFVI includes compute, network and storage resources that are virtualized.
- **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.

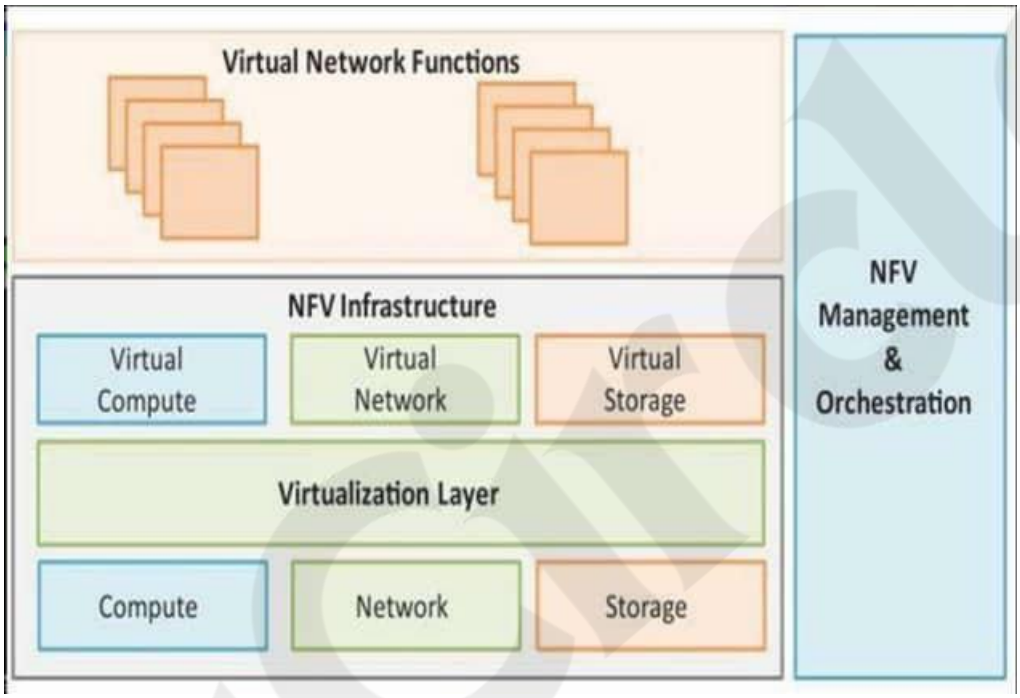


Figure 3.11: NFV architecture

NFV (Network Function Virtualization) runs network tasks using software on virtual/cloud servers instead of physical devices. It separates software from hardware, making updates and testing easier without changing the hardware. This reduces both equipment cost and power usage. Because it's cloud-based, many users or services can share the same virtual resources. NFV is used for managing data and control plane operations in fixed and mobile networks.

Figure 3.11 shows the NFV architecture, as being standardized by the European Telecommunications Standards Institute (ETSI) [82]. Key elements of the NFV architecture are as follows:

- **Virtualized Network Function (VNF):** VNF is a software implementation of a network function which is capable of running over the NFV Infrastructure (NFVI).
- **NFV Infrastructure (NFVI):** NFVI includes compute, network and storage resources that are virtualized.
- **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.

- NFV runs network functions in software on virtualized cloud resources, separate from physical hardware.
- It allows easy upgrades, reduces equipment and power costs, and supports sharing across multiple services.
- NFV is used only for data plane and control plane functions in fixed and mobile networks.

NFV can be used to virtualize home networks by using a Home Gateway that provides Wide Area Network (WAN) connectivity for services like the Internet, IPTV, and VoIP as shown in the figure 3.12, This Home Gateway performs several important functions such as acting as a DHCP server, handling Network Address Translation (NAT), functioning as a firewall, and enabling application-specific routing. It assigns private IP addresses to devices in the home, translates them to a public IP using NAT, and provides specific routing for applications like VoIP and IPTV.

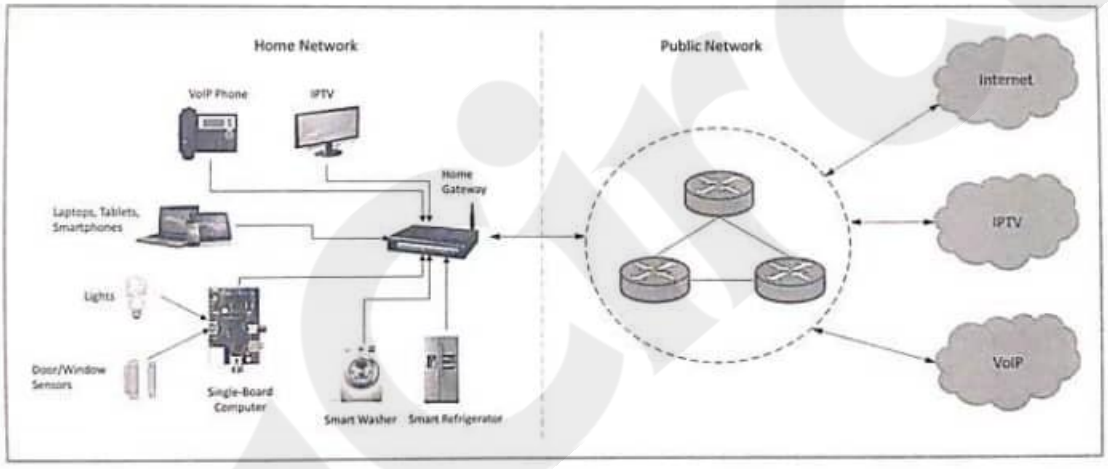


Figure 3.12: Conventional home network architecture

Figure 3.13 shows how NFV can be used to virtualize the Home Gateway. The NFV infrastructure in the cloud hosts a virtualized Home Gateway. The virtualized gateway provides private IP addresses to the devices in the home. The virtualized gateway also connects to network services such as VoIP and IPTV.

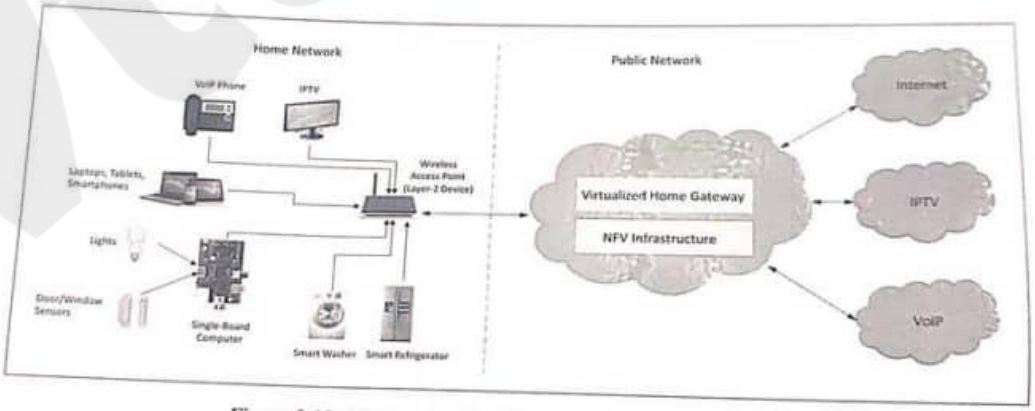


Figure 3.13: Home network with virtualized Home Gateway

2.5 Need for IoT Systems Management

IoT systems are complex and often involve many devices that need to be efficiently managed. The main reasons for IoT systems management are:

1. **Automating Configuration:**
   Automates the setup of IoT devices to ensure consistency and reduce manual errors.
   This is especially useful when managing large numbers of devices or nodes.
2. **Monitoring Operational & Statistical Data:**
   Collects runtime and performance data like CPU and memory usage.
   Helps diagnose faults and monitor overall system health.
3. **Improved Reliability:**
   Validates configurations before applying them to ensure system stability.
   Prevents issues caused by incorrect or untested settings.
4. **System Wide Configuration:**
   Applies configuration changes to all devices at once to avoid mismatches.
   Ensures reliable operation using an 'all or nothing' approach.
5. **Multiple System Configurations:**
   Some systems may need different valid configurations for different times or conditions.
   This flexibility helps adapt to varying operational requirements.
6. **Retrieving & Reusing Configurations:**
   Management systems can fetch and reuse existing configurations across similar devices.
   This ensures consistent setup, especially when adding new devices to the IoT system.

2.6 Simple Network Management Protocol (SNMP):

SNMP is a widely used protocol for monitoring and configuring network devices like routers and switches. Figure c shows the components o the entities involved in managing a device with SNMP.
It involves components like the Network Management Station (NMS), Managed Device, Management Information Base (MIB), and SNMP Agent.
NMS sends commands to manage devices using SNMP, while MIB stores device data in structured form using Object Identifiers (OIDs).
SNMP operates at the application layer and uses UDP as its transport protocol.

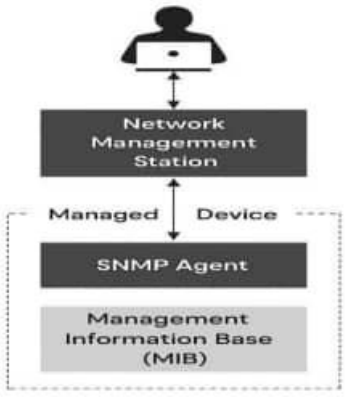


Figure 4.1 managing a device with SNMP

Limitations of SNMP

1. **Stateless Nature:**
   SNMP is stateless, so the application must handle the entire logic of configuration and error recovery.
   It requires maintaining state and intelligence for consistent device management.
2. **Connectionless Protocol:**
   SNMP uses UDP, which does not confirm receipt of messages.
   This makes communication unreliable, especially for critical configurations.
3. **Lack of Writable Objects in MIBs:**
   Many MIBs don't support writing, limiting SNMP to read-only tasks.
   Without writable data, it can't perform actual device configuration.
4. **Difficulty in Data Differentiation:**
   SNMP doesn't clearly separate configuration data from state data.
   This makes managing and interpreting data in MIBs more complex.
5. **Configuration Retrieval Issue**:
   SNMP makes it hard to retrieve and restore current device configurations.
   It lacks efficient support for configuration playback and retrieval.
6. **Security Concerns in Early Versions**:
   Older SNMP versions lacked strong security, leaving data exposed to threats.
   Later versions improved security but made SNMP more complex to use.

2.7 Network Operator Requirements

To improve network management protocols, the Internet Architecture Board (IAB) and IETF organized a 2002 workshop with network operators and protocol developers. Based on operator feedback, a list of key network operator requirements was compiled to guide future developments. List as follows

1. **Ease of use:** from the operators point of view, ease of use is the key requirement for any network management technology.
2. **Distinction Between Configuration and State Data:** Operators need a clear separation: configuration data changes system behaviour, while state data reflects operational status.
3. **Fetch Data Separately:** Configuration and state data should be fetched independently, especially when comparing devices.
4. **Configure the Network as a Whole:** Support is required for configuring the entire network at once, not device-by-device.
5. **Configuration Transactions Across Devices:** Enable configuration changes across multiple devices as a unified transaction.
6. **Configuration Deltas:** Systems should support generating minimal changes (deltas) between configuration states.
7. **Dump and Restore Configurations:** Ability to export (dump) and import (restore) configuration settings is essential.
8. **Configuration Validation:** It should be possible to check if a configuration is valid before applying it.

9. **Configuration Database Schemas:** A standard format for configuration data models and schemas is needed across operators.
10. **Comparing Configurations:** Devices should maintain order and format, making it easier to compare versions using tools like `diff`.
11. **Role-Based Access Control (RBAC):** Implement access based on user roles to limit access rights to necessary tasks only.
12. **Consistency of Access Control Lists (ACLs):** Ensure ACLs are consistent across all devices through verification tools.
13. **Multiple Configuration Sets:** Support for different versions of configurations (active vs candidate) should be available.
14. **Support for Data-Oriented & Task-Oriented Access Control:** Both data-focused (like SNMP) and task-focused (like CLI) access models should be supported.

2.9 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.

Figure 4.2 shows the layered architecture of NETCONF protocol.

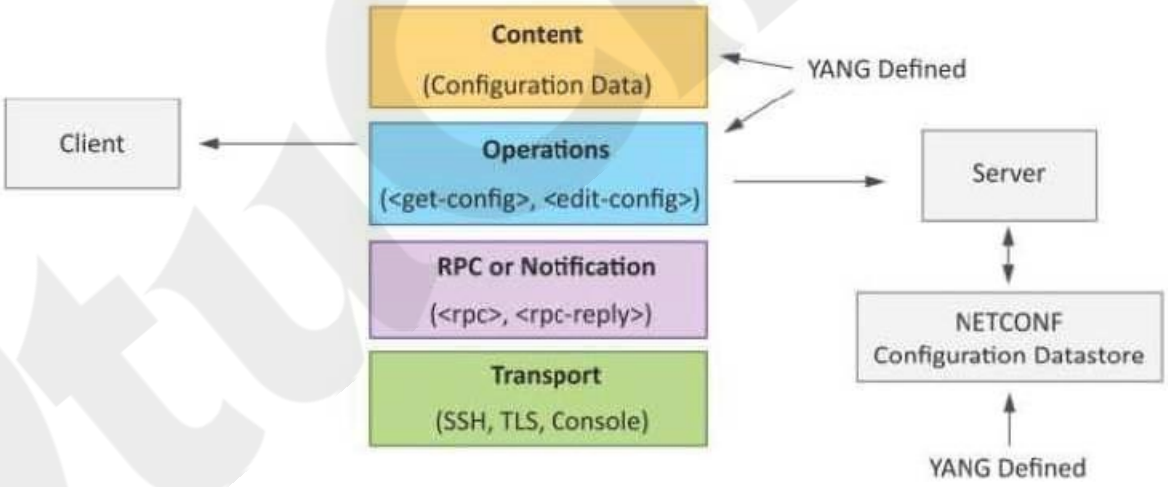


Figure 4.2 NETCONF protocol layers

- NETCONF works on SSH (Secure SHell transport layer protocol) transport protocol.
- Transport layer provides end-to-end connectivity and ensures 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. Table 4.1 provides a list of some commonly used NETCONF operations.

- 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 modelling 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.

Operation	Description
connect	Connect to a NETCONF server
get	Retrieve the running configuration and state information
get-config	Retrieve all or a portion of a configuration datastore
edit-config	Loads all or part of a specified configuration to the specified target configuration
copy-config	Create or replace an entire target configuration datastore with a complete source configuration
delete-config	Delete the contents of a configuration datastore
lock	Lock a configuration datastore for exclusive edits by a client
unlock	Release the lock on a configuration datastore
get-schema	This operation is used to retrieve a schema from the NETCONF server
commit	Commit the candidate configuration as the device's new current configuration
close-session	Gracefully terminate a NETCONF session
kill-session	Forcefully terminate a NETCONF session

Table 4.1: List of commonly used NETCONF RPC methods

2.10 YANG

YANG is a data modelling 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 define 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 Defines four types of nodes for data modelling as shown in table 4.2

Node Type	Description
Leaf Nodes	Contains simple data structures such as an integer or a string. Leaf has exactly one value of a particular type and no child nodes.
Leaf-List Nodes	Is a sequence of leaf nodes with exactly one value of a particular type per leaf.
Container Nodes	Used to group related nodes in a subtree. A container has only child nodes and no value. A container may contain any number of child nodes of any type (including leafs, lists, containers, and leaf-lists).
List Nodes	Defines a sequence of list entries. Each entry is like a structure or a record instance, and is uniquely identified by the values of its key leafs. A list can define multiple key leafs and may contain any number of child nodes of any type.

Table 4.2: YANG Node Types

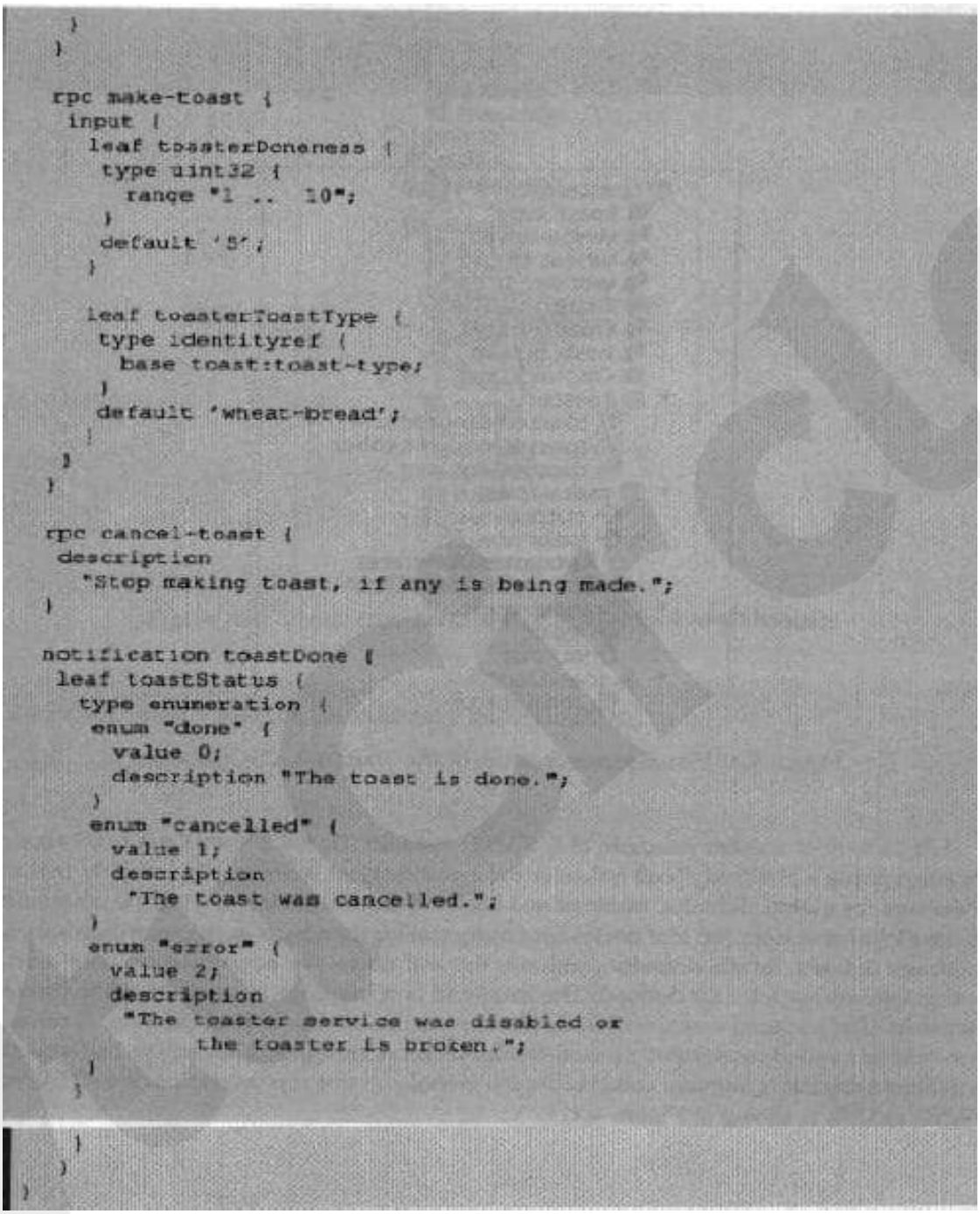
YANG Module Example 1:

- Let us now look at an example of a YANG module. Box 4.1 shows a YANG module for a "network-enabled toaster".
- 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').
- A tree representation of the toaster YANG module is shown in figure 4.3

Box 4.1: YANG version of the Toaster-MIB

```
module toaster {

  yang-version 1;

  namespace
   "http://example.com/ns/toaster";

  prefix toast;
```

```
revision "2009-11-20" {
 description
   "Toaster module";
}

identity toast-type {
 description
   "Base for all bread types";
}

identity white-bread {
 base toast:toast-type;
}

identity wheat-bread {
 base toast-type;
}

identity wonder-bread {
 base toast-type;
}

identity frozen-waffle {
 base toast-type;
}

identity frozen-bagel {
 base toast-type;
}

identity hash-brown {
 base toast-type;
}

typedef DisplayString {
 type string {
   length "0 .. 255";
 }
}

container toaster {
 presence
   "Indicates the toaster service is available";
 description
   "Top-level container for all toaster database objects.";
 leaf toasterManufacturer {
   type DisplayString;
   config false;
   mandatory true;
 }

 leaf toasterModelNumber {
   type DisplayString;
   config false;
   mandatory true;
 }

 leaf toasterStatus {
   type enumeration {
    enum "up" {
      value 1;
    }
    enum "down" {
      value 2;
    }
   }
   config false;
   mandatory true;
```



```
    }
  }

  rpc make-toast {
   input {
    leaf toasterDoneness {
     type uint32 {
      range "1 .. 10";
     }
     default '5';
    }

    leaf toasterToastType {
     type identityref {
      base toast:toast-type;
     }
     default 'wheat-bread';
    }
   }
  }

  rpc cancel-toast {
   description
    "Stop making toast, if any is being made.";
  }

  notification toastDone {
   leaf toastStatus {
    type enumeration {
     enum "done" {
      value 0;
      description "The toast is done.";
     }
     enum "cancelled" {
      value 1;
      description
       "The toast was cancelled.";
     }
     enum "error" {
      value 2;
      description
       "The toaster service was disabled or
            the toaster is broken.";
     }
    }
   }

    }
   }
  }
```

Figure 4.3: Visual representation of the Toaster YANG Module

YANG Module Example 2:

- Let us now look at an example of a YANG module. Box 4.2 shows a YANG module for configuring a HAProxy load balancer for a commercial website.
- It has four main parts: global, defaults, frontend, and backend.
- The **global** part sets values like max connections and mode.
- The **defaults** part sets things like retries and timeout.
- The **frontend** part sets port bindings.
- The **backend** part defines which servers will handle the load.
- A reusable structure called **server-list** is used to list the servers.
- This structure is created using something called a **grouping**.
- A tree diagram of this module is shown in Figure 4.4.

Box 4.2: YANG Module for HAProxy configuration

```
module haproxy {
  yang-version 1;

  namespace "http://example.com/ns/haproxy";

  prefix haproxy;

  import ietf-inet-types { prefix inet; }

  description
  "YANG version of the ";

  revision "2014-06-01" {
    description
```

```
      "HAProxy module";
   }

container haproxy {
   description
   "configuration parameters for a HAProxy load balancer";

      container global {
         leaf maxconn {
         type uint32;
         default 4096;
         }

         leaf mode {
         type string;
         default 'daemon';
         }
      }

      container defaults {
         leaf mode {
         type string;
         default 'http';
         }

         leaf option {
         type string;
         default 'redispatch';
         }

         leaf retries {
         type uint32;
         default 3;
         }

         leaf contimeout {
         type uint32;
         default 5000;
         }

         leaf clitimeout {
         type uint32;
         default 50000;
         }
```

```
        leaf srvtimeout {
        type uint32;
        default 50000;
        }
    }

    container frontend {

        leaf name {
        type string;
        default 'http-in';
        }

        leaf bind {
        type string;
        default '*:80';
        }

        leaf default_backend {
        type string;
        default 'webfarm';
        }
    }

    container backend {
        leaf balance   {
        type string;
        default 'roundrobin';
        }
        uses server-list;
    }
}

    grouping server-list {
    description "List of servers to load balance";
        list servers {
        key address;
        leaf address {
          type inet:ip-address;
        }

        leaf maxconn {
          type uint32;
          default 255;
        }

        }
    }
}
```

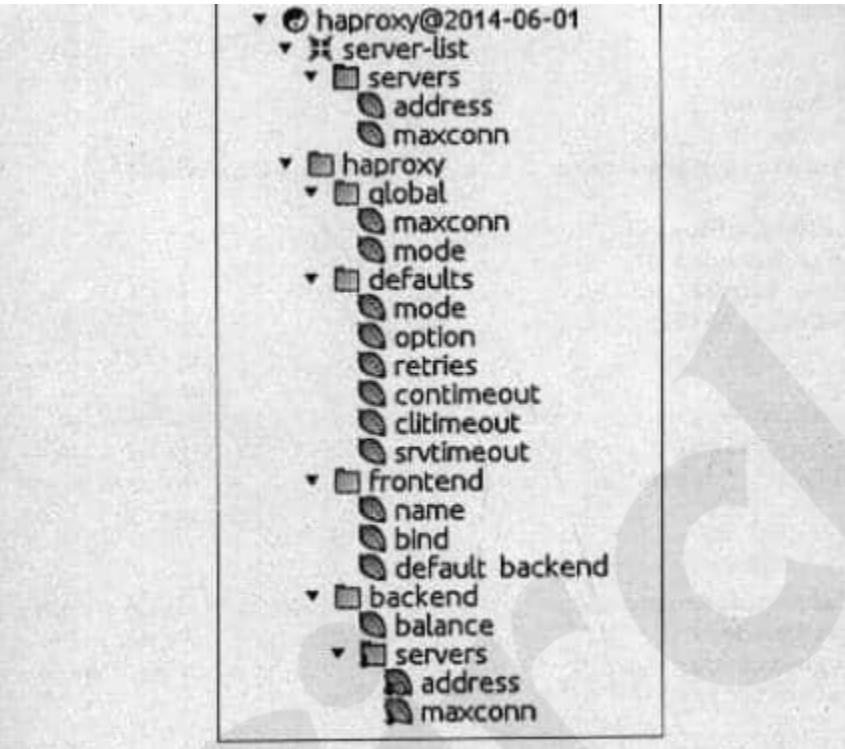


Figure 4.4: Visual representation of HAProxy YANG Module

2.11 IoT Systems Management with NETCONF-YANG.

In this section you will learn how to manage IoT systems with NETCONF and YANG.

Figure 4.5 shows the generic approach of IoT device management with NETCONF-YANG. Let us look at the roles of the various components:

1. **Management System**
   - Used by the operator to send NETCONF messages to configure IoT devices.
   - Also receives notifications and current state information from the device as NETCONF messages.
2. **Management API**
   - Allows management applications to perform NETCONF operations programmatically.
   - Supports starting NETCONF sessions, reading and writing configuration data, reading state data, and invoking RPCs—just like an operator.
3. **Transaction Manager**
   - Responsible for executing all NETCONF transactions.
   - Ensures that all ACID properties are followed:
     - **Atomicity**: A transaction is either fully completed or not done at all.
     - **Consistency**: Ensures the device moves from one valid configuration state to another.
     - **Isolation**: Guarantees that even if transactions run at the same time, the final state is as if they ran one after another.

- **Durability**: Once a transaction is committed, the changes will remain even after system failures.

4. **Rollback Manager**
   - Generates necessary steps to revert the current configuration back to its original or previous state.
   - Used when a configuration error occurs or needs to be undone.
5. **Data Model Manager**
   - Keeps track of all YANG data models used for device configuration.
   - Also monitors which applications provide specific data parts for each YANG model.
6. **Configuration Validator**
   - Checks whether the configuration after applying a transaction is valid.
   - Prevents incorrect or invalid configurations from being saved or applied.
7. **Configuration Database**
   - Stores both the **configuration data** (intended settings) and **operational data** (actual working state of the device).
   - Acts as the central storage for all device-related data.
8. **Configuration API**
   - Allows applications on the IoT device to:
     - Read configuration data from the configuration datastore.
     - Write operational data to the operational datastore.
9. **Data Provider API**:
   Applications on the IoT device can register for callbacks for various events using the Data Provider API. It also allows the applications to report statistics and operational data.

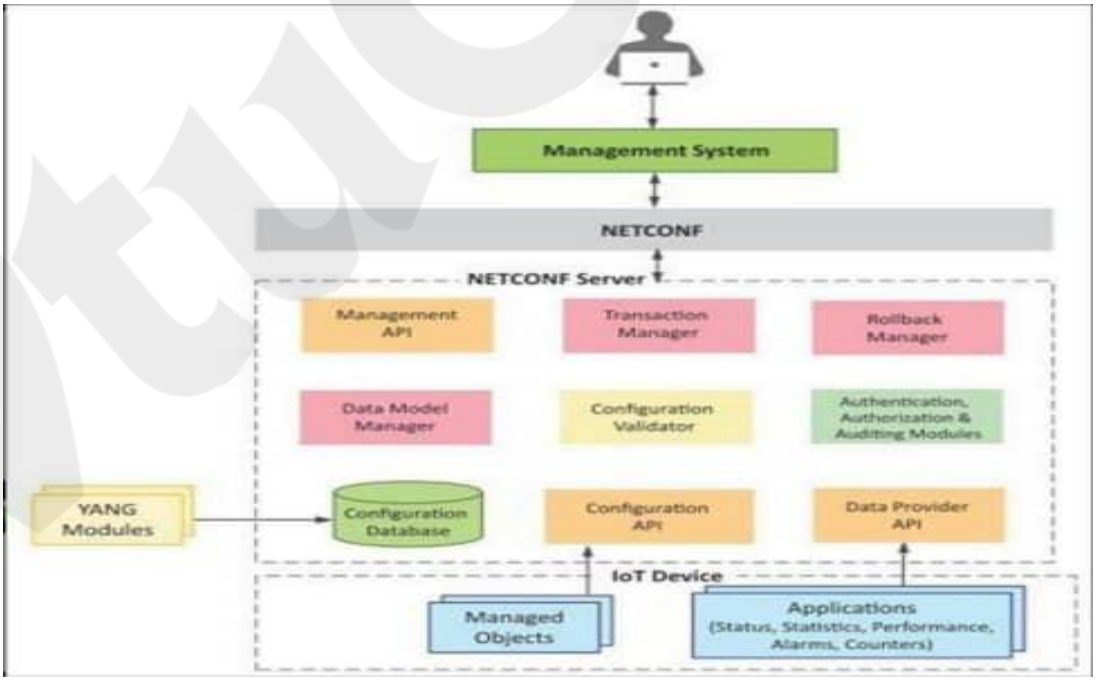


Figure 4.5: IoT device management with NETCONF-YANG -a generic approach

2.11.1 : NETOPEER

While the previous section described a generic approach of IoT device management with NETCONF-YANG, this section describes a specific implementation based on the Netopeer tools. Netopeer is a set of open sources NETCONF tools built on the Libnetconf.
Figure 4.6 shows how to manage an IoT device using the Netopeer tools.

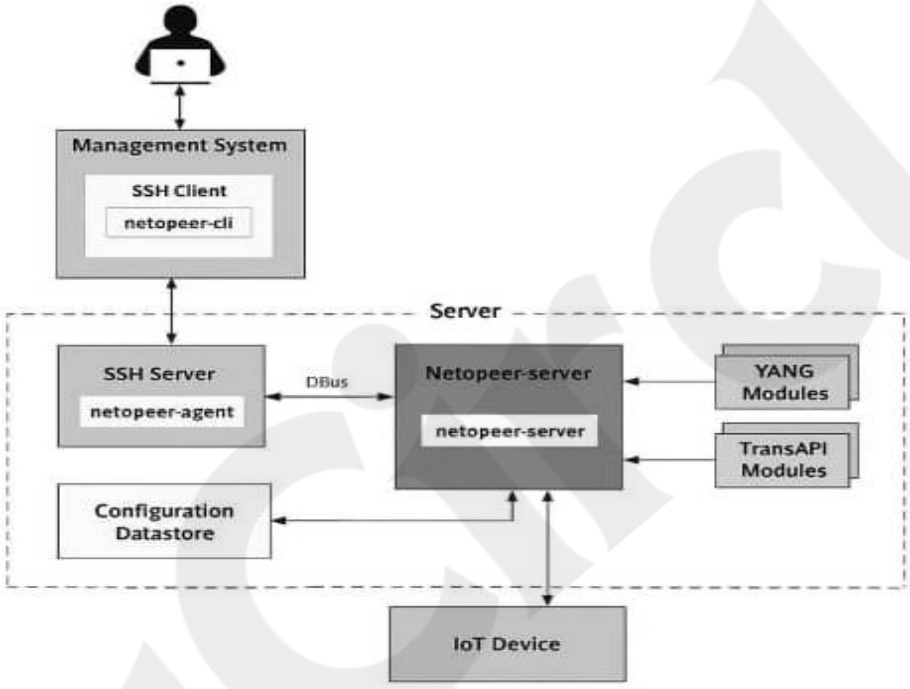


Figure 4.6: IoT device management with NETCONF - a specific approach based on Netopeer tools

The Netopeer tools include:

1. **Netopeer-server**: Netopeer-server is a NETCONF protocol server that runs on the managed device. Netopeer-server provides an environment for configuring the device using NETCONF RPC operations and also retrieving the state data from the device.
2. **Netopeer-agent**: Netopeer-agent is the NETCONF protocol agent running as a SSH/TLS subsystem. Netopeer-agent accepts incoming NETCONF connection and passes the NETCONF RPC operations received from the NETCONF client to the Netopeer-server.
3. **Netopeer-cli**: Netopeer-cli is a NETCONF client that provides a command line interface for interacting with the Netopeer-server. The operator can use the Netopeer-cli from the management system to send NETCONF RPC operations for configuring the device and retrieving the state information.
4. **Netopeer-manager**: Netopeer-manager allows managing the YANG and Libnetconf Transaction API (TransAPI) modules on the Netopeer-server. With Netopeer-

manager, modules can be loaded or removed from the server.

5. **Netopeer-configurator**: Netopeer-configurator is a tool that can be used to configure the Netopeer-server.

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. **Compile the YANG model with the 'lnctool'** which comes with Libnetconf. Libnetconf provides a framework called Transaction API (TransAPI) that provides a mechanism of reflecting the changes in the configuration file in the actual device. The 'lnctool' generates a TransAPI module (callbacks C file). Whenever a change is made in the configuration file using the NETCONF operations, the corresponding callback function is called. The callback functions contain the code for making the changes on the device.
3. **Fill in the IoT device management code** in the TransAPI module (callbacks C file). This file includes configuration callbacks, RPC callbacks, and state data callbacks.
4. **Build the callbacks C file** to generate the library file (.so).
5. **Load the YANG module** (containing the data definitions) and the TransAPI module (.so binary) into the Netopeer server using the Netopeer manager tool.
6. **The operator can now connect** from the management system to the Netopeer server using the Netopeer CLI.
7. **Operator can issue NETCONF commands** from the Netopeer CLI. Commands can be used to change the configuration data, get operational data, or execute an RPC on the IoT device.

Module 2 Question Bank

1. Which communication protocols are used for M2M local area networks?
2. What are the differences between Machines in M2M and Things in IoT?
3. How do data collection and analysis approaches differ in M2M and IoT?
4. What are the differences between SDN and NFV?
5. Describe how SDN can be used for various levels of IoT?
6. What is the function of a centralized network controller in SDN?
7. Describe how NFV can be used for virtualizing IoT devices?
8. Why is network wide configuration important for IoT systems with multiple nodes?
9. Which limitations make SNMP unsuitable for IoT systems?
10. What is the difference between configuration and state data?

11. What is the role of a NETCONF server?

12. What is the function of a data model manager?

13. Describe the roles of YANG and TransAPI modules in device management?