Enabling interoperability amongst global IoT networks using shared-infrastructural models

Easwaran, K.I.

Award date:
2018

Technische Universiteit Eindhoven University of Technology
System Architecture and Networking

Enabling Interoperability Amongst Global IoT Networks using Shared-Infrastructural Models

Master Thesis

Krishna Iyer Easwaran

Supervisor

Dr. Rudolf H. Mak
System Architecture and Networking
Technische Universitiet Eindhoven

Thesis Committee

Dr. Rudolf H. Mak
System Architecture and Networking
Technische Universitiet Eindhoven

Johan Stokking, MSc
The Things Industries B.V.

Dr. Tom Verhoeff
Mathematics and Computer Science
Technische Universitiet Eindhoven

Abstract

In the field of large scale Internet of Things(IoT) networks using the LoRaWAN protocol, The Things Network(TTN) is a very popular global, crowd-funded, decentralized network operating on the principle of openness. TTN has experienced an exponential growth in popularity with more than 47000 developers, 4500 active gateways, routing close to 6 million messages every day (as of the writing of this document). The Things Network has two variants i.e., a public community network offered free of charge to its users and commercial networks for industrial customers.

However, the current iteration of The Things Network stack (ttnV2) and the process of deploying it for customers has two major points of concern. First, the process of deploying a dedicated instance for each customer and maintaining a separate instance for the public community is not scalable with regards to installation, management and modification. Second, there is presently no mechanism to exchange traffic between these instances which results in a fragmented ecosystem.

This thesis demonstrates how, using architectural description techniques, two or more architectures can be compared for relative advantages. This provides meaningful insights into potential issues in the architectural design of the new stack. It also identifies how the Software-as-a-Service (SaaS) design paradigm offers solutions to scalability issues in the microservice based TTN software stack, by proposing deployment schemes that attempt to balance various (often opposing) requirements. This project also evaluates different state-of-the-art tools such as Kubernetes that facilitate these deployment schemes. Furthermore, the project demonstrates how the TTN instances can seamlessly exchange traffic with each other, by the design and implementation of a Publish-Subscribe based system called the Peering Broker, and provides arguments as to why this design is more scalable compared to peering mechanisms provided by the LoRaWAN specifications. And finally, this project also demonstrates how the exchange of traffic between networks can be quantified to facilitate a fair and transparent exchange.

Contents

List of Figures

List of Tables

Acknowledgements

”It is the mark of a good action that it appears inevitable in retrospect (R.L.Stevenson)” and in retrospect, it is my duty to recognize the actions of those without whom this project would simply not have materialized. The first of these is the continued support and guidance of my thesis supervisor Dr. Rudolf H. Mak (System Architecture and Networking group, Technische Universteit Eindhoven), whose detailed inputs into the construction of this document as well as the execution of this thesis are immensely treasured. The second is the patience and the backing of my company supervisor Johan Stokking (co-founder and CTO, The Things Industries), whose guidance was indispensable in both my comprehension of the necessary technological aspects and the subsequent implementation of the project itself. Next, the constant support and encouragement from my colleges Hylke Visser (Lead Stack Engineer, TTI) and Eric Gourlaouen (Backend Developer, TTI) and the endless positivity from Wienke Giezeman (CEO and Co-Founder, TTI) continues to inspire me to improve myself in all my endeavors.
I would like to take this opportunity to thank EIT Digital Masters school for believing in my abilities and providing me an opportunity to take part in this wonderful masters program.
There is no one else on this planet that I owe a deeper debt of gratitude than to my family who continue to be the reason I push myself to greater heights.
And finally, I would like to thank my friend Reshma Emilien Alva for extending a helping hand when I needed it the most, and to my best friend Despina Stefanoska for all her persistent useful support and her endless useless memes.

Krishna Iyer Easwaran,
25 Aug 2018,
Amsterdam, Netherlands

1 Introduction

The ever-increasing need for connectivity has driven technology to new heights. Smartphones and their complimentary wireless network infrastructure have placed the internet and all its features at our disposal, for better or for worse. The next logical phase of this technology is the Internet of Things (IoT), defined as "the domain that is working to connect devices to communicate with each other without human intervention". Traditionally, the construction and maintenance of wireless network infrastructure is dominated by big corporations, usually establishing monopoly over large regions and imposing their own terms on their customers. To prevent monopoly and centralization in the IoT domain, The Things Network (TTN) was created as a free, open source, distributed IoT network solution that could be created by anyone, anywhere . The technology is an open IoT Network leveraging LoRaWAN (a long range low power Internet of things protocol) that promotes decentralization and interoperability between a multitude of devices and networks. These networks are usually paid for and maintained by multiple parties including crowd-sourced, public networks making the reach global and the possibilities endless. Presently, TTN is based on a dual licensing model with the free, open source public networks and on-demand custom private networks, supported by the Things Industries (TTI), the commercial entity that provides the solutions (including software) for the TTN.

Though the concept of open, decentralized IoT networks is a great step to prevent monopoly and promote interoperability, the existing version of TTN (ttnV2) has its drawbacks. Firstly, the separation of the public and private network deployments does not scale well. It is necessary to separately install, configure, monitor and maintain each private network. This creates numerous individual points of failure due to its localized approach. Secondly, this model is geographically restrictive and creates redundancy as the same geographical region may be served by networks from multiple customers. These restrictions lead to the fact that data cannot be exchanged/peered between different networks, which defeats the vision of global interoperability. And finally, individuals and businesses who host public networks need to be compensated when their resources are used by other customers.

In order to achieve true decentralization and interoperability and to solve the aforementioned problems, TTN is being redesigned to ease its usage as a SaaS (Software as a Service) platform and to allow peering (or the exchange of data between networks).

In this project, we explore how the redesigned software stack supports new requirements, how peering can be realized for IoT networks with large throughput to allow seamless exchange of traffic between them, how SaaS-based tools and techniques enable scalability and ease maintenance and finally how the exchange of data between networks can be quantified.

Consequently, the rest of this introductory chapter is devoted to broadly setting the context and stating the problems that will be elaborated through the rest of the document.

1.1 Context and background

Before stating the problem, it is essential to describe the company, its product and its processes in order to contextualize the problem. The rest of this section is dedicated to this description. The concepts and terms broadly outlined here will be examined in more detail in subsequent chapters.

1.1.1 The Things Network

The Things Network identifies itself as "a global, crowd-sourced, open, free and decentralized Internet of Things Network."[1]. It is an eco-system of IoT devices and software and a vibrant global community driven by a collective ideology amongst its users and developers, aiming to be mutually beneficial.

The Things Network comprises of two symbiotic entities, specifically:

- The Things Network Foundation (TTN):
  A non-profit foundation dedicated to providing software and support to create communities of geographically localized networks that are initiated and maintained by local communal efforts. The foundation operates on the principles of transparency and decentralization while the communities are expected to adhere to the The Things Network manifesto [2] and honor the principles of Fair-Use.

- The Things Industries B.V. (TTI):
  A commercial entity that, in addition to developing the software and providing the necessary support to The Things Network Foundation, develops and distributes customized software and services to industrial customers. A major part of the revenue derived by this entity is invested in maintaining the The Things Network Foundation (the services of which are free of cost to the community members).

The seamless coordination between the two entities preserves the ecosystem and ensures its continuity.

1.1.2 The Product

TTI develops and maintains a specific software implementation of the LoRaWAN network server (to be explored in detail in the next chapter), which is a middle-ware that provides routing services between IoT devices placed in the field and the software applications that use the data from the devices. Presently, TTI uses version 2 (herein referred to as ttnV2) of its software stack which is currently used for both the public community network and private commercial networks. It is quite interesting to note that 90% of the software stack is open-source and is open for developers to modify and improve. As a matter of fact, the backbone of the TTN Ecosystem is the active developer community who provide bug-fixes and feature updates as well as serving as lead users for new features. TTI offers this software along with a few closed-source proprietary components, installation and operational support and makes it available as a premium package to industrial clients. TTI also supports certain customer-specific features in these installations that are not available in the public network.

1.1.3 The Process

As mentioned earlier, the software stack that is developed by TTI (i.e., ttnV2) needs to be installed onto machines (either real or virtual) and maintained for proper functioning. Though this process will be dealt with in depth in a subsequent chapter, it is vital to understand the diversity of these installations and their consequence. A few instances of the ttnV2 software stack are installed for public use on certain virtual machines, each instance serving a particular region (ex: ttn_eu, ttn_us etc). These installations are called *Public Community Networks* and are hosted and maintained free of cost to the users. TTI offers its commercial customers similar installtions of the software, either on a cloud VM or on customer defined machines for a pre-determined price. The process of installing and configuring these networks to each customer's requirements is referred to as a *Deployment*. Currently, these installations are what is referred to as *Single tenant installations* as each instance serves only one tenant (customer). Such a process cannot scale as the number of customers increase for reasons that will become apparent in subsequent chapters.

1.1.4 Shortcomings

Every system has its deficiencies and the current TTN eco-system is no exception. The deficiencies that are relevant to this project are listed below.

- The ttnV2 software stack is not extensible to include new requirements such as updated LoRaWAN specifications (Ex: LoRaWAN 1.1).

- The lack of peering (data-sharing) between the private networks and the between private-public networks causes global isolation and redundancy as it is geographically restrictive for each customer.
- The manual deployment and operation model is tedious to maintain and does not scale with an increase in the number of customers.
- There is no mechanism to defray the gateway owners who contribute traffic to the network, which curtails the mass adoption of TTN networks globally.

Please note that each item in this list will be elaborated in dedicated chapters subsequently in this document.

1.2 Problem statement

The aforementioned deficiencies produce the following requirements for this project:

1. Identify and evaluate more efficient deployment schemes and evaluate tools that enable them.
2. Design and implement a mechanism for peering between the networks to share traffic.
3. Identify and/or design a scheme to monitor and log metrics that can determine value provided by gateway owners to the network.
4. Analyse and document TTN's new architecture to gain better insight into the TTN ecosystem and to identify potential flaws (and suggest improvements).

1.3 Document Structure

The following table provides a global overview of the structure of this document.

Table 1: Overview of this document

Chap.	Main Theme	Research Space	Implementation Space
2	Understanding LoRaWAN	Explain essential concepts of LoRaWAN required for the rest of this document.	-
3	Analysis of the V3 Architecture	Study of LoRaWAN based IoT stacks, comparative study of TTN v2 and v3 architectures, Motivation for newer architecture	Document new stack using Arch description techniques
4	Deployment	Global Network sharing considerations, Multi-tenancy, Load distribution, scalability	Evaluation of tools, PoC Implementation of Shared deployments with ease of scalability and load balancing using tools found
5	Collaboration between Networks	Routing and Peering, Metering of Traffic, Packet Delivery strategies, Discovery and Handoff considerations	Discovery, Authentication, Secure Data Routing to the correct end point. [Opt] Optimal Routing strategies
6	Conclusions	Summarize the project and analyze the outcome	-

2 Understanding LoRaWAN

This chapter briefs the reader through aspects of LoRaWAN that are necessary to understand the rest of this document. Those who are already quite familiar with LoRaWAN may skip ahead to the next chapter.

2.1 Overview

LoRa[3] is a long-range, low-power radio modulation protocol that is part of the LPWAN(Long Range, Low Power Wide Area Network) class of modulation. LoRaWAN (LoRa Wide Area Network) is a MAC (Medium Access Control) layer protocol designed on top of the LoRa modulation scheme which is governed by the LoRa Alliance The following image explains the LoRaWAN Software stack:

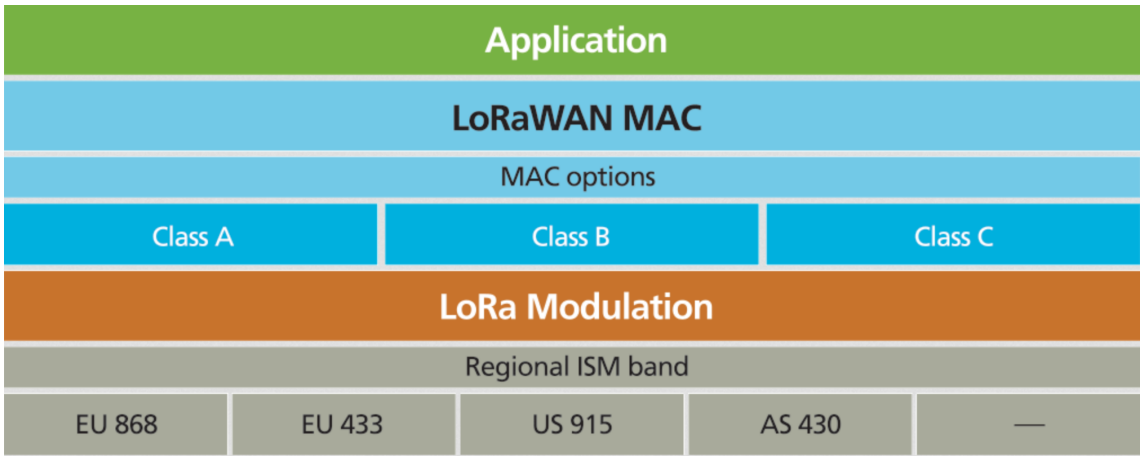

Figure 1: LoRa Stack
Source: [3]

When compared to the *OSI model*, LoRaWAN defines only the following layers:

LoRaWAN Specification	ISO Model Specification
LoRa Radio Modulation	Physical Layer
LoRaWAN MAC Protocol	Data Link Layer
Application	Covers the upper OSI layers namely Network, Transport, Session, Presentation and Application.

Table 2: Comparison of OSI and LoRaWAN Software stack.

The *Application* layer is left to the user to implement as required by individual use cases.
The following image shows a typical LoRaWAN End-to-End Installation:

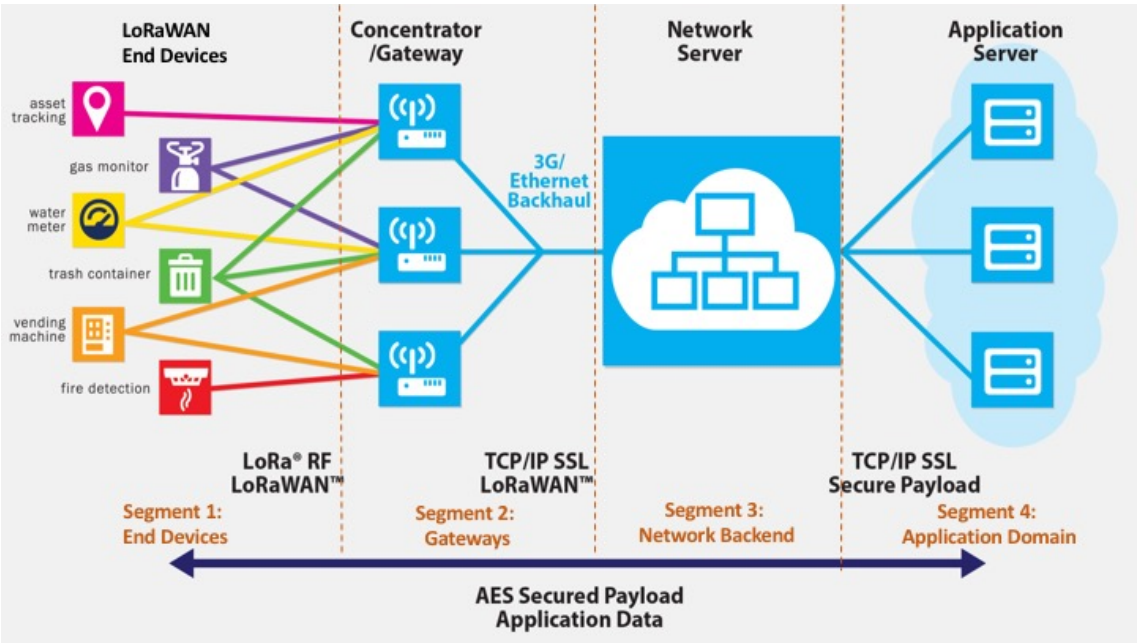

Figure 2: Typical LoRaWAN End-to-End Installation
Source: [3]

2.2 Key terms

The standard implementation of the LoRaWAN network consists of the following parts:

- **Device (End-Device)**:
  A LoRaWAN-enabled piece of hardware that usually consists of some sensors and (in some cases) actuators, that is deployed in various physical environments, usually to gather some sensor data. The most important requirements for these devices are that they must be able to communicate over long ranges (10km or more) and be able to operate with extremely low power, often sustaining themselves on battery power for years.

- **Application**:
  This is a software program that uses the data produced by the devices. Applications and end devices are usually designed and developed together to cater to specific use cases.

- **Gateways**:
  Gateways are specialized devices that have LoRaWAN and IP capabilities. They convert data between LoRaWAN and IP-based protocol formats. Since end devices are low powered, they transmit whenever data is available. This means that the Gateways must always be listening to incoming traffic from the end devices. Most gateways handle only one downlink (refer below) at at time due to power supply constraints and hence, downlinks must be ”scheduled” on a gateway by an upstream entity. It is interesting to note that devices can work with any network but gateways must be configured to only one network at any given time.

- **Network Server/Backend**:
  This is a *generic* term for the hosted software that routes data between Devices and Applications. The network server also performs functions such as gateway/device management, De-duplication[1] of the same data from multiple gateways and encryption/decryption of MAC data.

[1] In the field, the same uplink packet may be forwarded by different gateways to the Network Server, which then

- **Application Server**:
  This is a hosted server that stores and processes the data from the devices and makes it available for the application. This is introduced so that the applications are not required to be listening to the network at all times but can query the application server periodically for the data.
  The application Server and the Network server together form the *Backend* of the LoRaWAN network.

- **Device Identification**:
  Each Device that supports LoRaWAN is assigned a special Extended Unique Identifier; DevEUI which is expected to be *unique* globally. However, to allow a device to freely connect and re-connect to any network, the Network Server stores a relative identifier called the Device Address (DevAddr) to which this DevEUI is mapped.

- **Uplink**: An uplink is the transmission of a Data Packet *from the End Device to the Backend.* Uplink Data frames have a *frame-counter* that is used in tandem with the DevAddr to make them uniquely identifiable.

- **Downlink**: A downlink is the transmission of a Data Packet *from the Backend to the End Device.* Downlink need not be synchronous to Uplinks.

2.3 Important Design Considerations

While designing a system meant for Low Power Devices in real-world applications, there are a number of factors that need to be considered. The most important of these factors are discussed here, with focus on how they influence the design of LoRaWAN.

- **Power restriction on End Devices and Gateways**:
  The *Transmission (Tx) Power* and *Duty Cycle* restriction on the LoRa Radio Modulation are the most important constraints on LoRaWAN network. In order to meet regional regulations [4] to operate in the unregulated ISM (Industrial, Scientific and Medical) Band, restrictions are placed on the maximum transmission power and Maximum On Time (Max Duty Cycle) of the LoRa Radio. Hence, the LoRaWAN network is designed such that only simple functions such as device connection and transfer of uplink and downlink data are performed on the LoRaWAN network and all the other complex functionality is delegated to the IP-based Network Server.

- **Topology**:
  Most medium to large scale IoT device deployments are meant for use-cases where real-world data is periodically read and relayed to the applications for processing/decision making. This property of these networks eliminates the need for the End Devices to communicate amongst each other. As a result, the LoRaWAN Network is deployed in a *hierarchal star topology* where a large number of devices communicate with a single gateway[2] whereas a large number of gateways are in-turn managed by a single Network Server.

- **Security**:
  LoRaWAN uses encryption on Application Data so that only the intended applications can use them. In addition, the MAC layer administration information is also encrypted to secure the LoRaWAN radio behaviour. So, LoRaWAN employs a two-level encryption scheme with the Application payload encrypted using Application Keys (shared between devices

uses the DevAddr and the frame-counter to keep track the metadata of all the uplink frames and forward only one copy of the application data to the Application Server. This process is referred to as *De-duplication.*

[2]Though, it must be noted that since duplicate uplink packets may arrive via different gateways at the network server, this is not a simple 1:n star network.

and the Application Server only) and the whole network data encrypted using Network Keys(shared between the devices and the Network Server only). Both are AES-128 symmetric keys. This also means that application servers can freely connect to any network server and be sure that the data is secure on any network.

2.4 A note on the Specifications

As described in an earlier section, LoRaWAN provides a technical descriptions for the Physical (Radio), MAC and Application layers. The v1.1 [5] specification is actually composed of three documents namely:

1. LoRaWAN Network Specification (1.1):
   This discusses the MAC layer and the new Join procedures including the generation of the keys used by the various parts of the system.

2. LoRaWAN Backend Interfaces (1.0):
   This document defines the design for Roaming between two networks and will be explored in detail in the section on Peering.

3. LoRaWAN Regional Parameters (1.0.3):
   This document discussed the various frequency bands and physical layer parameters such as Tx Power and Air-time for different regions across the world. For the most part, the contents of this document are beyond the scope of this thesis.

We explore the LoRaWAN Network Specification (1.1) and discuss the aspects that are most relevant for this thesis.

2.5 Identifiers

The LoRaWAN Network Specification (1.1) specification defines and describes a list of *Identifiers* that represent devices/networks/data that are a part of the network. The following table describes them briefly:

Table 3: List of Keys Used in LoRaWAN 1.1

Identifer	Length (bits)	Purpose
DevEUI	64	This is a globally Unique value that identifies a specific device
JoinEUI	64	This is a globally Unique value that identifies a Server(Join) where this device is registed to. This is used to register a device to a network.
NETID	24	A Unique identifier that is assigned to a Network which is purchased by the Network Operator from the LoRa alliance.
DevAddr	32	An ID that is assigned to a device once it is registered to a network. This ID is temporary and can be reassigned. A portion of the DevAddr is the NetID. This is similar to the concept of a Dynamic IP address.

2.6 Key Handling

Security is one of the most important features of LoRaWAN which includes encryption and integrity checks. There are two levels of encryption namely MAC layer encryption and Application layer encryption. The keys used for this purpose and their details are found in the table below.

Key	Type	Function
AppKey	RootKey	Used to derive the Application Session Keys
NwkKey	RootKey	Used to derive the Derive the Network Session Keys
JSEncKey	Derived Key	Device: Encrypt Join Request, decrypt Join accept, Network: Encrypt Join Accept, decrypt Join Request
JSIntKey	Derived Key	Used by the Network to check the integrity of the Join Request
FNwkSIntKey	Derived Key	Used to perform integrity checks during Roaming (Explained later)
SNwkSIntKey	Derived Key	Used to perform integrity checks during Roaming (Explained later)
NwkSEncKey	Derived Key	Used to Encrypt Network Payloads
AppSKey	Derived Key	Used to encrypt Application Payloads

Note: All the keys mentioned above are AES-128 symmetric keys.

2.7 Summary

In this chapter, we looked at some key concepts of LoRaWAN including its architecture, important entities in a typical installation, design considerations, specification documents, identifiers and encryption keys. Knowledge of the above concepts should be sufficient to read through the rest of this document. Now that this groundwork is established, in the next chapter, we look at the first part of this project, i.e, analysis of the ttnV3 stack.

3 Analysis of the V3 Architecture

With the fundamentals of LoRaWAN in place, we now commence the discussion of the first part of this project. TTI redesigned their network stack with a newer architecture and are currently in the process of realizing this in software. However, it is crucial to analyze this new architecture, understand the motivation behind this redesign and compare it with other LoRaWAN based stacks. This serves two main purposes; identification of potential flaws and detailed documentation of the stack using standard architectural description methods, both of which are dealt with in this chapter.

3.1 Problem Context

The Things Network V2 stack is a *middleware* that is constructed using the microservices architecture[6], which is a design paradigm that advocates the decomposition of large functional software monoliths into smaller, inter-connected functional components (services). The various services that ttnV2 is comprised of is shown in the image below.

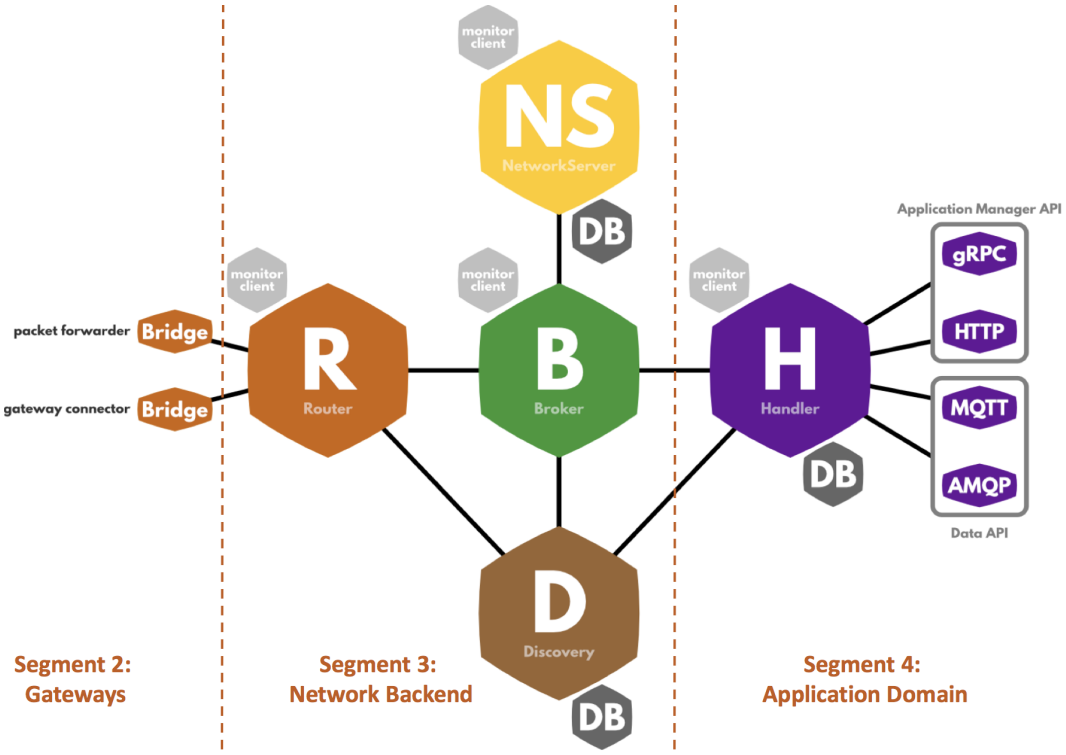

Figure 3: TTN-V2 Backend structure
Source: [7]

Please note that in the above diagram, *Segment 1: End Devices* (as described in the chapter on LoRaWAN) is not explicitly displayed but it does indeed exist as part of the TTN ecosystem.
Each service performs a set of vital functions that are listed in the table below:

Table 4: Key functions of microservices in ttnV2

Service	Key Responsibilities
[LoRaWAN] Gateway	A device that performs protocol translation between LoRaWAN and IP (Internet Protocol)
Bridge	A service that does protocol translation between the gateway's legacy UDP protocol running on most LoRa gateways and the Protocol Buffers over gRPC are used in TTN
Router	Handles all gateway related functions such as uplink, downlink management, gateway scheduling and storing gateway metadata.
Broker	Decides if a device is to be served or not based on its Device Address, Deduplicates Application packets and gateway metadata, forwards uplinks to handler, and chooses the best downlink path.
Network Server	Maintains device state and MAC layer configuration, Checks message integrity checks (MICs) for downlinks.
Discovery Server	A server through which components find each other (follows Service Oriented Architecture Principle).
Handler	Handles all Application related functions including encrypting/decrypting application payloads, providing a server for Applications to subscribe to the Uplinks.
Application	A user defined Software components that is the destination of all uplinks and the source of all downlinks.

TTI discovered certain short-comings in the ttnV2 stack that triggered a complete re-design of their architecture (and an inevitable re-implementation). In order to explain these short-comings, the description presented above needs further elaboration, which is done in the first part of this chapter.

3.2 Requirements

The requirements of this chapter are as follows:

- Explore the ttnV2 stack to understand the motivation for the newer architecture.
- Analyze and document the ttnV3 stack using architectural description methods.
- Compare the two architectures to identify potential issues in ttnV3.
- Explore the architecture of other LoRaWAN based networks and contrast them with TTN.

3.3 The Current (ttnV2) stack

The functioning of the ttnV2 stack is best explained with an example use case. Firstly, consider a temperature sensor equipped with a LoRaWAN Radio. The values read by this device are of interest to an application. This device is placed at a suitable location where the temperature is to be measured. This device joins the network(TTN) using its Device ID or DevEUI (Device Extended Unique Identifier) which is used to identify this device. The device sends this DevEUI as part of the network join message, depicted in the fig 4.

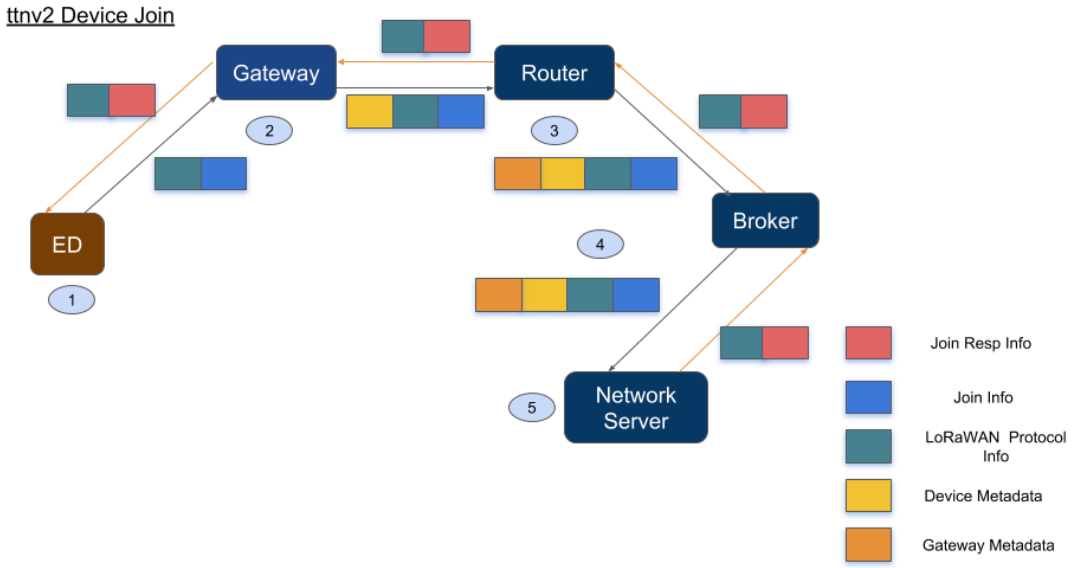

Figure 4: Join Flow in ttnv2

Table 5: Explanation of ttnV2 Device Join Procedure

No	Component	Action
1	End Device	The Temp sensor value (application data) is read, packaged into LoRaWAN packets (Uplink Messages) and transmitted over LoRa Radio Modulation.
2	Gateway	The uplink messages from the devices are demodulated from LoRa and packaged as IP packets along with device meta-data such as RSSI (Received Signal Strength Index) and SNR (Signal to NoiseRatio of the received signal).
3	Router	The IP packets are routed to the correct broker based on the Device Address(DevAddr)* of the End Device as each broker handles packets only from a specific list of devices.
4	Broker	Since multiple gateways may forward the same uplink message, the broker de-duplicates these messages by retaining all the metadata and forwarding only one copy of the actual application data. In Addition, the Uplink integrity is verified by the Broker.”
5	Network Server	The Network Server maps the devices to the application and returns this mapping when requested by the broker so that the uplink packet is sent to the correct application.
6	Handler	The end-to-end encrypted application data is decrypted here and converted to a format required by the application
7	Application	The application is a program that is the final recipient of the application data (Temperature Value).

Once the device has joined the network, it sends temperature value as Uplink messages which is then processed by the TTN backend and forwarded to the the application as described in the image below:

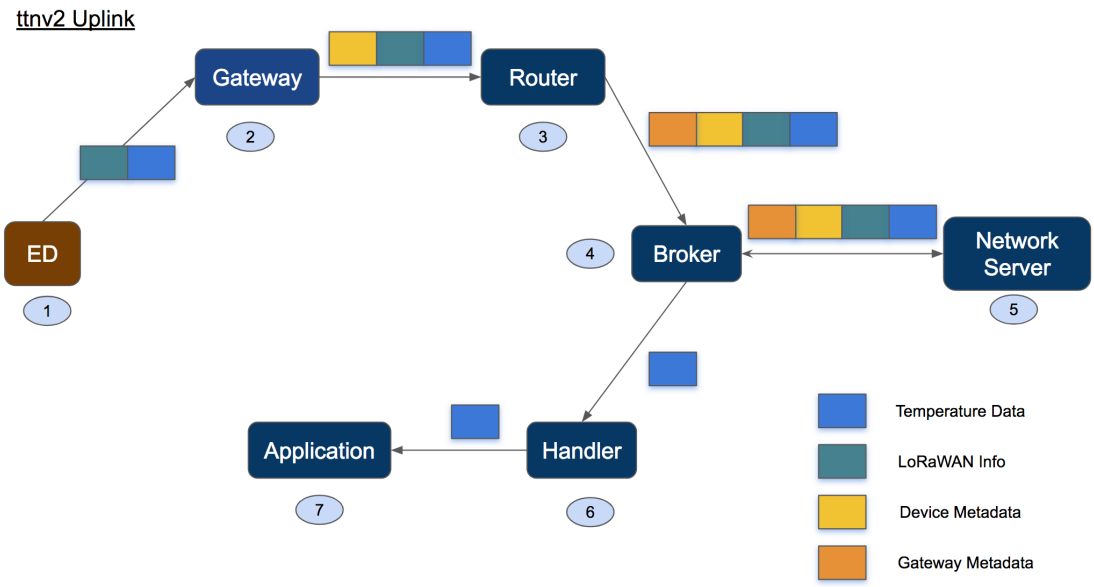

Figure 5: Uplink Flow in ttnv2

ED refers to the End Device, in this case, the temperature sensor. The various steps depicted in the above image is explained in the table below:

Table 6: Steps involved in routing an Uplink Message in ttnV2

No	Component	Action
1	End Device	The Temp sensor value(application data) is read packaged into LoRaWAN packets (Uplink Messages) and transmitted over LoRa Radio Modulation.
2	Gateway	The uplink messages from the devices are demodulated from LoRa and packaged as IP packets along with device meta-data such as RSSI(Received Signal Strength Index) and SNR(Signal to Noise Ratio of the received signal).
3	Router	The IP packets are routed to the correct broker based on the Device Address(DevAddr)* of the End Device as each broker handles packets only from a specific list of devices.
4	Broker	Since multiple gateways may forward the same uplink message, the broker de-duplicates these messages by retaining all the metadata and forwarding only one copy of the actual application data. In Addition, the Uplink integrity is verified by the Broker.
5	Network Server	The Network Server maps the devices to the application and returns this mapping when requested by the broker so that the uplink packet is sent to the correct device.
6	Handler	The end-to-end encrypted application data is decrypted here and converted to a format required by the application.
7	Application	The application is a program that is the final recipient of the application data (Temperature Value).

Now, let us consider another case where a device capable of some actuation (Ex: An automatic lighting system). In this case, the User's Application triggers a Downlink message to the End Device. The various steps involved in delivering the Downlink message are shown in Table 6:

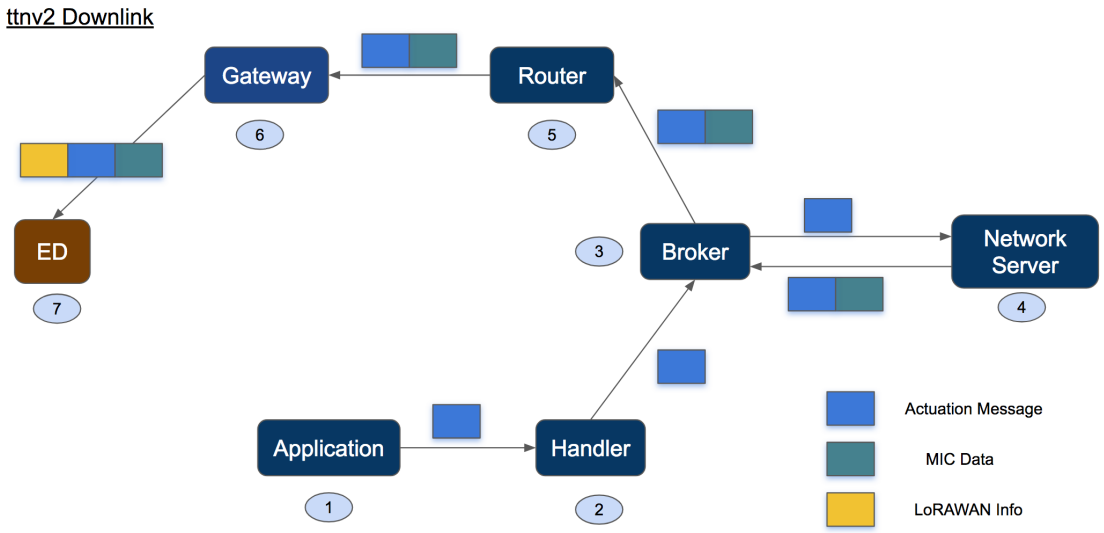

Figure 6: Downlink Flow in ttnv2

Please refer to the table 7 for an explanation of the steps.

Table 7: Steps involved in routing a downlink in ttnV2

No	Component	Action
1	Application	The application needs to send an actuation message to the device. It sends this message to the handler along with the DevEUI of the target device.
2	Handler	The Handler encrypts the actuation message after performing a format conversion if necessary. It then passes on this message to the broker.
3	Broker	The Broker forwards the downlink message to the Network Server for integrity calculation. Once it receives the updated downlink message from the Network Server, it forwards it to the router.
4	Network Server	The Network Server updates the Device state and generates the Message Integrity Code (MIC) which will be used by the device to check the downlink integrity. This is then added to the downlink message. Then, this data is sent back to the broker.
5	Router	Most gateways can hold only one downlink message at a time. Hence, the router buffers the downlink until it can pass it on to a gateway that is free.
6	Gateway	The gateway transmits the downlink message over LoRa radio to the End Device during the Rx cycle of the device. Each device has two receive windows defined in the specification.
7	End Device	The end device receives the downlink, decrypts it, checks its integrity and takes appropriate action (Ex: actuation)

3.3.1 Motivation for a new Architecture

The Things Industries (TTI) made a decision to redesign their network stack to satisfy the following requirements:

- Adoption of LW1.1:
  The LoRa alliance released a newer version of its specifications namely , LoRaWAN1.1[5]

which introduced the concept of Roaming (exchanging data between Networks), modified the Network design to incorporate a designated server to maintain keys (Join Server), added more flexibility in the MAC layer (padding of MAC commands) and proposed the inclusion of Wireless bands for more regions. TTI decided to redesign their Network stack to incorporate all these changes while at the same time providing backwards compatibility to older LoRaWAN versions within the same Network stack.

- Introduction of Peering:
  One of the major changes in LoRaWAN 1.1 is the Roaming feature. This allows networks to exchange traffic between each other to cover broader regions and make LoRaWAN more accessible to its users. The ttnv2 stack was not designed for this feature and hence needed a new design and the introduction of Peering mechanisms to realize Roaming.

- Facilitation of SaaS based Hosting:
  In order to scale up the number of customers served and to ease the burden of managing the networks for these customers, TTI decided to move from a distributed (per-customer) to a shared deployment scheme. The ttnv2 stack is not capable of taking full advantage of a shared infrastructure and hence needs a re-design.

- Address design issues in V2:
  The ttnv2 was designed as an SOA(Service Oriented Architecture), where all components/services had to register with the Discovery Server and used it to connect to each other. This resulted in the discovery server being a Single Point Of Failure (SPOF) that caused the entire system to fail with it. This needed to be overcome by using a network clustering and inter-network discovery strategies which are incorporated in ttnV3. In addition, some bugs that could not be fixed in ttnV2 due to the volume of the change involved, are fixed in ttnV3.

3.4 Analysis of Other LoRaWAN based stacks

Most LoRaWAN based stacks that are currently active are commercial and closed-source. A notable exception is the LoRaServer, an open-source *standard* implementation of the LoRaWAN Network Backend created with the intention of serving as an evaluation server. The LoRaServer implementation uses a monolithical architecture, that makes it interesting to analyze. Its architecture is described in the image below:

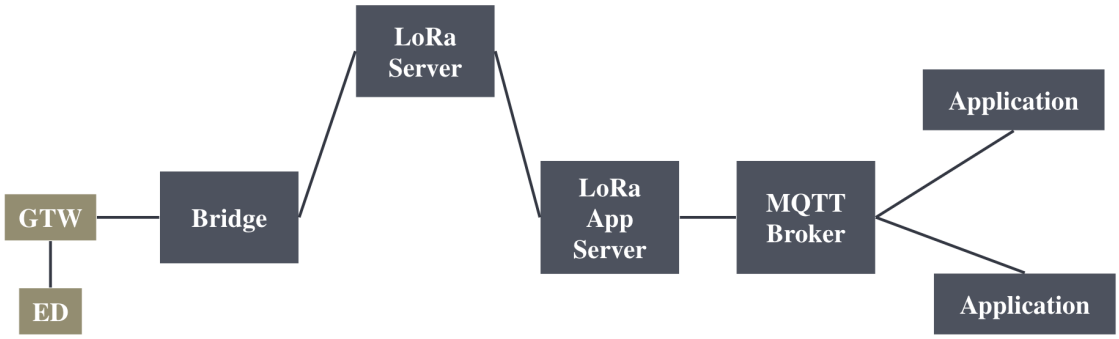

Figure 7: Architectural Overview of the LoRaServer

In the above figure, it is interesting to note the following things:

- There is a Bridge component that connects the gateway to the rest of the network.

- The entire network backend is a single service (LoRa Server); hence this architecture is referred to as a monolithical architecture.
- The LoRa App Server is analogous to the Handler in ttnV2, which handles all the functions related to applications.
- The LoRa Server supports only one method of connecting applications, via an MQTT broker. Applications need to listen to the MQTT broker on specific topics to get uplinks and publish to the same broker to push their downlinks.

3.5 Architectural Description of ttnV3

And now, we finally describe the architecture of ttnV3, starting with the overview, shown in the image below:

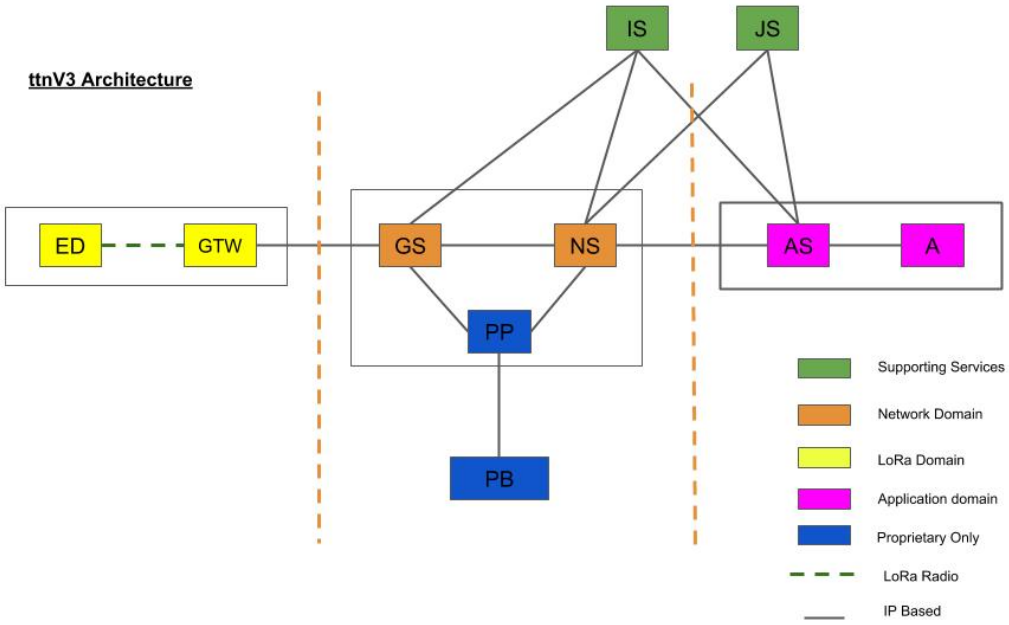

Figure 8: Architectural Overview of ttnV3

At first glance, certain aspects are evident in the newer architecture. With the exception of the *Network Server* component, all the others components have been replaced with alternatives. Also, new components have been introduced. The Join Server (JS) is one such component that is now responsible for securely maintaining the root keys of the devices. The Peering Broker (PB) (which is the focus of the next chapter) is introduced to enable exchange of traffic between networks. The PB is not deployed within the ttnV3 stack along with the rest of the components and hence, a Peering Proxy component is introduced as part of the stack to interact with the peering broker.
The key responsibilities of each of the components of the new architecture are described in the following table:

Table 8: Main Responsibilites of ttnV3 microservices

Service	Key Responsibilities
[LoraWAN] Gateway	A device that performs protocol translation between LoRaWAN and IP (Internet Protocol).
Gateway Server	Handles all gateway related functions such as uplink, downlink management, gateway scheduling and storing gateway metadata.
Network Server	Decides if a device is to be served or not based on its Device Address, Deduplicates Application packets and gateway metadata, forwards uplinks to the handler, and chooses the best downlink path. Maintains device state and MAC layer configuration, Checks message integrity checks(MICs) for downlinks.
Application Server	Handles all Application related functions including encrypting/decrypting application payloads, providing a server for Applications to subscribe to the Uplinks.
Join Server	A special server introduced in LoRaWAN 1.1 that is reponsible for Securely providing keys for encryption and integrity checks.
Identity Server	A server that manages users and their access to the Applications and Devices.
Application	A user defined Software components that is the recipient of all data uplinks and the initiator of all data downlinks.
Peering Proxy	A micro-service available only in the Proprietary version of the TTN stack responsible for inter-network peering.
Peering Broker	A regionally hosted message broker for inter-cluster peering.

3.5.1 Functional View

- **Join**:
  Fig 9 and table 9 explain the Join procedure for the V3 LoRaWAN 1.1 OTAA(Over the Air Activation) device join procedure:

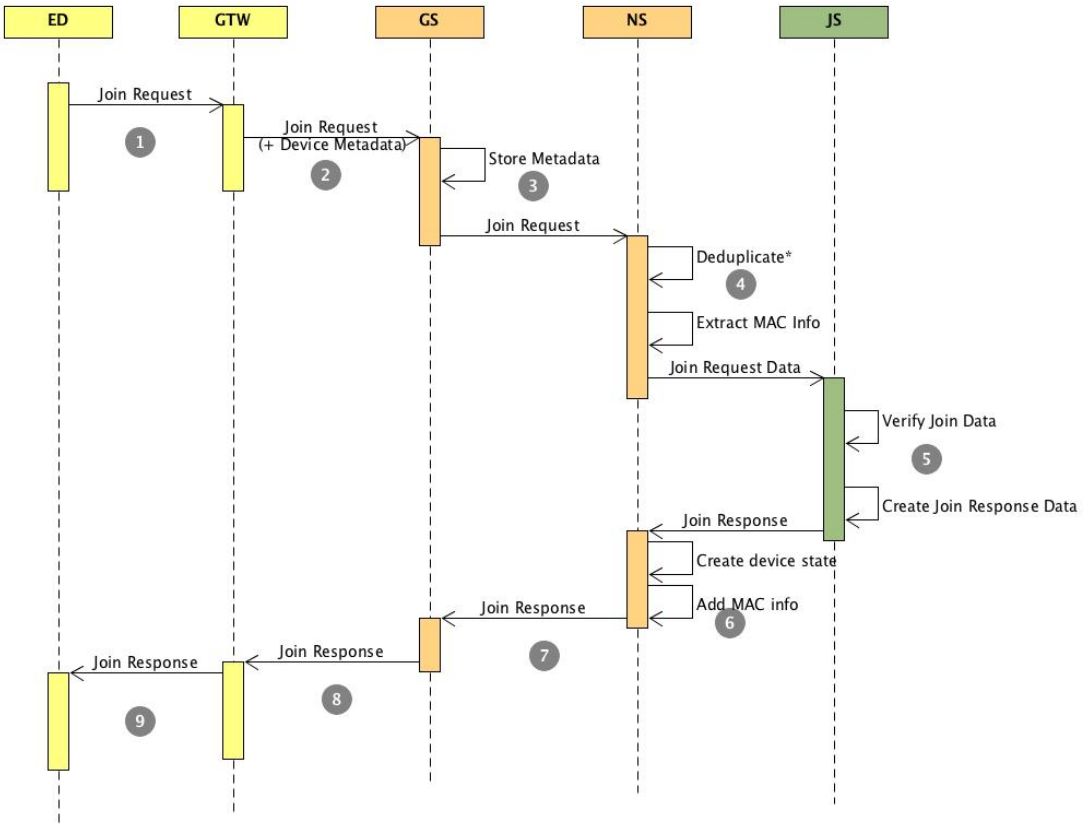

Figure 9: ttnv3 OTAA LW1.1 Join Sequence

[3]

Table 9: Join Sequence for TTNV3 (OTAA devices)

No	Action
1	Every End Device needs to send a Join Request to connect to a Network Server. The End Device sends it Device Identifier (DevEUI) and a Network Identifier (JoinEUI) in the Join Message. Both these are pre-programmed in the device.
2	The Gateway picks up the Uplink Message on the LoRa Radio, demodulates it to an IP packet and sends it to the Gateway Server, along with some metadata.
3	The Gateway Server is the first component of the ttnv3 stack. It picks up the uplink message from the Gateway and stores the device and gateway metadata. It then forwards the Uplink Message to the Network Server.
4	The Network server de-duplicates the message (if necessary) and filters out the MAC Info. Then it forwards the message to the Join Server.
5	The Join Server Verifies the DevEUI and JoinEUI and decides if this device would be served or not. This decision is communicated via the Join Response message. If the decision is positive, the Join Response contains the Device Address (DevAddr) assigned to this device, and the secret keys needed for communication.
6	The NS adds necessary MAC info to the Join Response and sends it to the GS.
7	The GS will schedule the downlink on a gateway.
8 & 9	The End device receives the Join Response via the gateway.

[3] De-duplication is not necessary in-case there are no duplicates.

- **Uplink**:
  Now we consider the same temperature sensor that we discussed earlier. The new network stack does not affect the operation of End Device. It still sends encrypted Uplink Messages to the gateway which now forwards it to the Gateway Server. The entire process is explained in the image below:

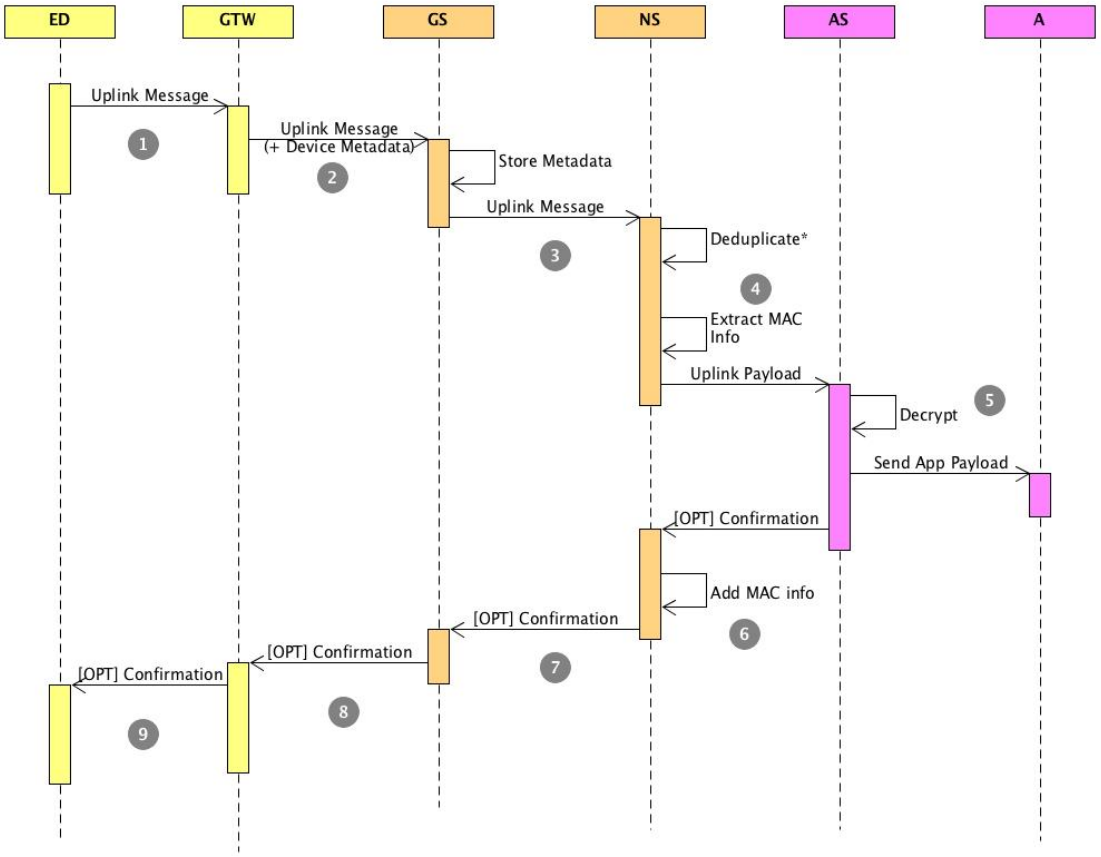

Figure 10: ttnv3 OTAA LW1.1 Uplink Sequence

Table 10: Steps involved in routing an Uplink in ttnV3

Step	Action
1	The End device sends an Uplink Message (Ex: Temperature Data) along with its Device Address. The Application data (Temperature) is secured by an application key(AppSKey) and the MAC info is secured by a Network Key(NwkSEncKey). This is broadcast for any gateway in its vicinity to receive.
2	The Gateway receives the Uplink Message on the LoRa Radio, demodulates it to an IP packet and sends it to the Gateway Server, along with some metadata.
3	”The Gateway Server picks up the uplink message from the Gateway and stores the device and gateway metadata. It then forwards the Uplink Message to the Network Server.
4	”The Network server de-duplicates the message (if necessary) and filters out the MAC Info. Then it forwards the message to the Application Server.
5	The Application Server then decrypts the Uplink message and sends it to the application.
6 to 9	The End device can optionally request a confirmation for the uplink which is then triggered by the AS and is sent via the NS, GS and finally the GTW to the device.

- **Downlink**:
  Similarly, the operation of the Users application remains unaffected by the new network stack. A downlink intended for a device (Ex: An actuating Device) is still initiated by the Application and Received by the Application Server. The sequence of handling the downlinks in the ttnv3 stack is shown in the image below:

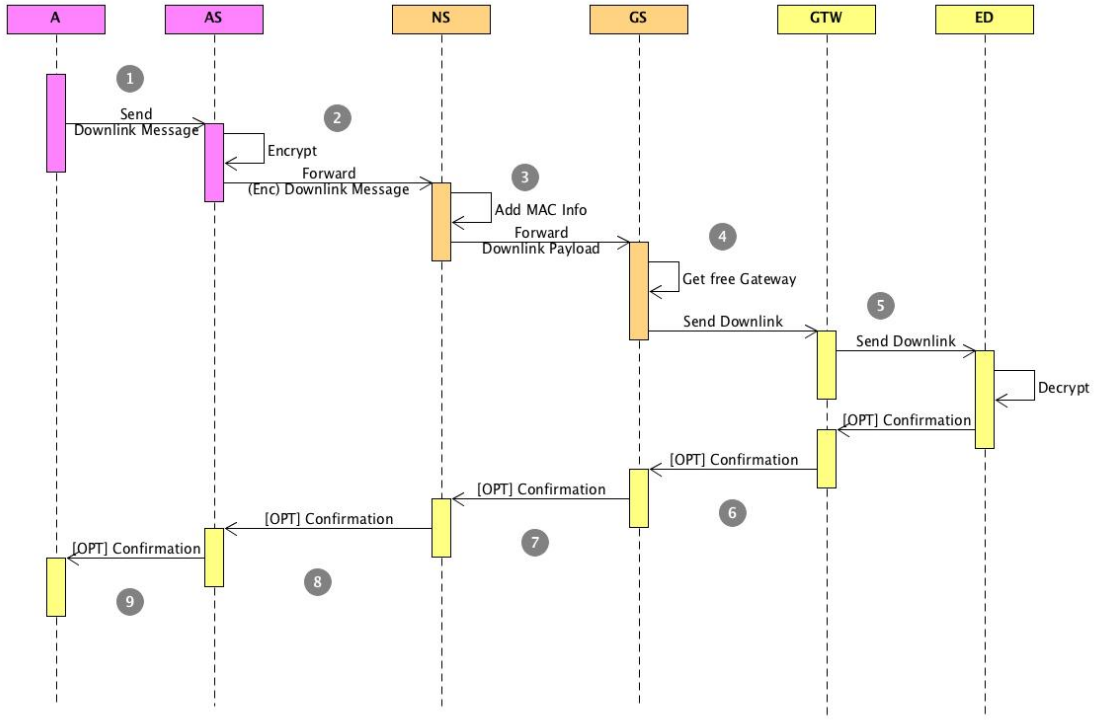

Figure 11: ttnv3 OTAA LW1.1 Downlink Sequence
Source: adapted from [5]

Table 11: Steps involved in routing a Downlink in the ttnV3 stack

Step	Action
1	All Downlinks are initiated by the application and a request is placed on the Application Server.
2	The application Server encrypts the message and forwards it to the network server.
3	The Network Server adds the necessary MAC Inforamation and forwards it to the Gateway Server.
4	The Gateway server then schedules the Downlink on an appropriate gateway.
5	The Gateway sends the Downlink message to the end device over LoRa Radio.
6 to 9	The Application may optionally request for a confirmation message that is initiated by the End Device and relayed across the network back to the Application.

Let us once again consider our example of a user with some temperature sensors installed in some location. The user (Ex: Web interfaces) registers the Application and the corresponding devices (temperature sensors) to an Application Server(AS) (either hosted by TTI or others). This Application Server needs to be connected to a Network Server(NS) that is operating in the region where the device is installed. This is usually done beforehand by the operators of the Application Server. This application now *subscribes* to uplinks from the device. The uplinks from the NS are now pushed to the AS which will either push it to the application or buffer it if necessary. The user can then login to the Application and access the temperature data.

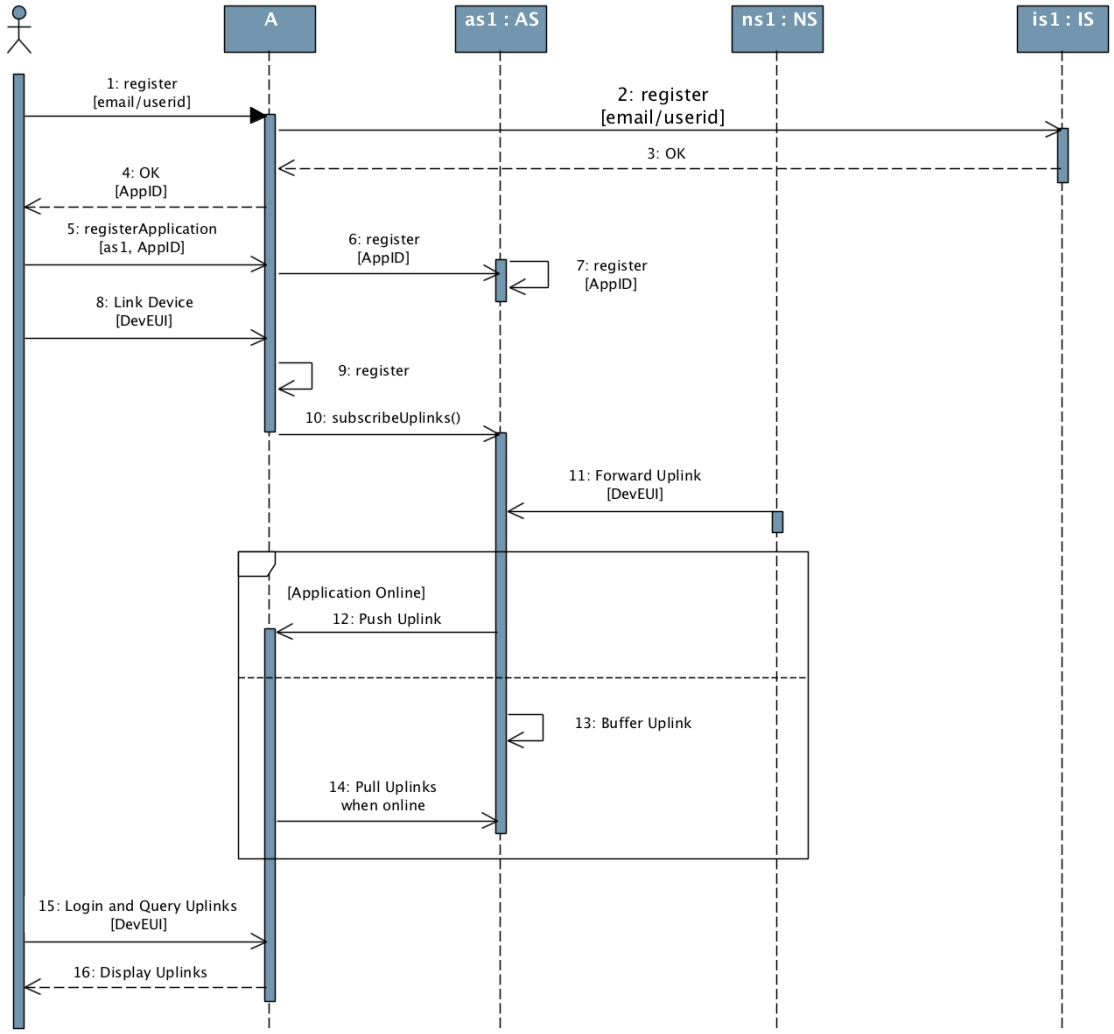

Figure 12: ttnv3 User Flow

3.6 Comparison of Responsibilities

The table 12 provides a concise comparison of the responsibilities of each of the components in ttnV2 and ttnV3 and the LoraServer.

Table 12: Comparison of responsibilities of micro-services in ttnV2 v/s ttnV3 v/s LoRa Server

Key Responsibilities	ttnV3	ttnV2	LoRa-Server
A device that performs protocol translation between LoRaWAN and IP(Internet Protocol).	[LoraWAN] Gateway	[LoraWAN] Gateway	[LoraWAN] Gateway
Handles all gateway related functions such as uplink, downlink management, gateway scheduling and storing gateway metadata.	Gateway Server	Router	LoRa Server
Decides if a device is to be served or not based on its Device Address, forwards uplinks to handler, and chooses the best downlink path.	Network Server	Broker	LoRa Server
Checks message integrity checks(MICs) for downlinks.	Network Server	v2 Network Server	LoRa Server
Maintains device state and MAC layer configuration	Network Server	v2 Network Server	LoRa Server
De-duplication of Uplinks	Network Server	v2 Network Server	LoRa Server
Handles all Application related functions including encrypting/decrypting application payloads.	Application Server	Handler	LoRa App Server
Buffers the uplinks for an application.	Application Server	Handler	LoRa App Server
Provides Keys for Encryption and Integrity checks.	Join Server	v2 Network Server	LoRa Server
A server that manages users and their access to the Applications and Devices.	Identity Server	Account Server	LoRa App Server
A user defined Software components that is the recipient of all data uplinks and the initiator of all data downlinks.	Application	Application	Application
A microservice available only in the Proprietary version of the TTN stack responsible for inter-network peering.	Peering Proxy	Not available	Not available
A regionally hosted message broker for inter-cluster peering.	Peering Broker	Not available	Not available

3.7 Summary

In this chapter, we explored three LoRaWAN based stacks with the objective of creating a plane of reference to analyze the ttnV3 stack. These stacks, namely the LoRa-Server, ttnV2 and ttnV3 itself are each designed with a different architectural pattern based on their requirements. The insights thus gained are listed below.

- The ttnV3 stack does not contain any single points of failure with the removal of the ttnV2 discovery server.
- Since the stack was designed with LW1.1 in mind, it can be easily extended to provide backwards compatibility to LW1.0 since LW1.0 is a subset of LW1.1
- The introduction of a special component to handle the exchange of traffic (the peering proxy) means that ttnV3 can support the LoRaWAN Device Roaming functions (explained in the next chapter).
- The detailed architectural and behavioral models documented in this chapter are a good reference point to understand the working of the newer architecture.

4 Deployment

Deployment is a vital phase of the software development life-cycle. It's defined as a list of steps performed in order to get software running and available to users. In this chapter, we discuss (in great depth) the current deployment process used by TTN and the issues that arise from it. We then explore alternative deployment schemes and the various architectural considerations that need to be made for those schemes. This results in a model architecture that balances some of the key concerns. We then analyze several tools that facilitate this model and test them for their functioning. And finally, we list some limitations and provide an outlook on future work in this regard.

4.1 Context

Deployment is a multi-step process involving software packaging, target preparation and installation and configuration. TTN currently employs a *single tenant, dedicated deployment* scheme, where each instance of the software stack (ttnv2) is dedicated to only one *tenant*(customer). There are public instances dedicated to the public community and, as of Aug 29 2018, about 25-30 single deployed instances each serving a private customer. All of these deployed instances are managed by TTI. The taxonomy of deployed instances is shown in figure 13.

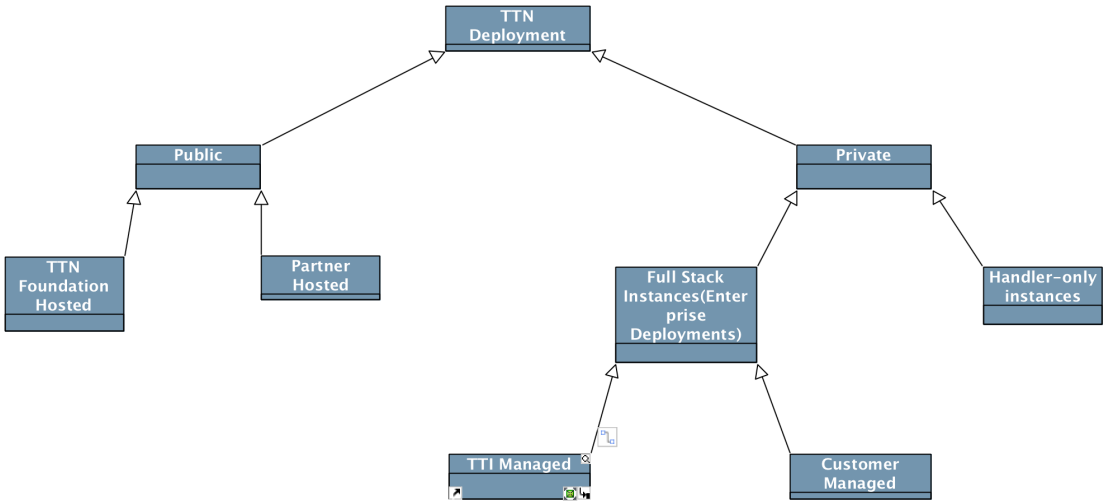

Figure 13: Taxonomy of Deployed Instances of the ttnV2 stack.

Note: Private handlers are special deployments where only the *Handler* component is privately managed but the rest of the network uses the public infrastructure.

4.1.1 Step 1: Pre-installation (Preparation)

This step is also referred to as *Software Packaging or containerization*. The micro-service design pattern advocates the decomposition of large software functions into logical sub-services (micro-services). These micro-services are then then packaged into *containers*. Containers are "*abstractions at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space*"[8]. This is different from Virtual Machines(VMs) as they are abstractions directly over the hardware layer with a separate operating system(guest OS) running on top of a host. The difference between the two is evident in the image below.

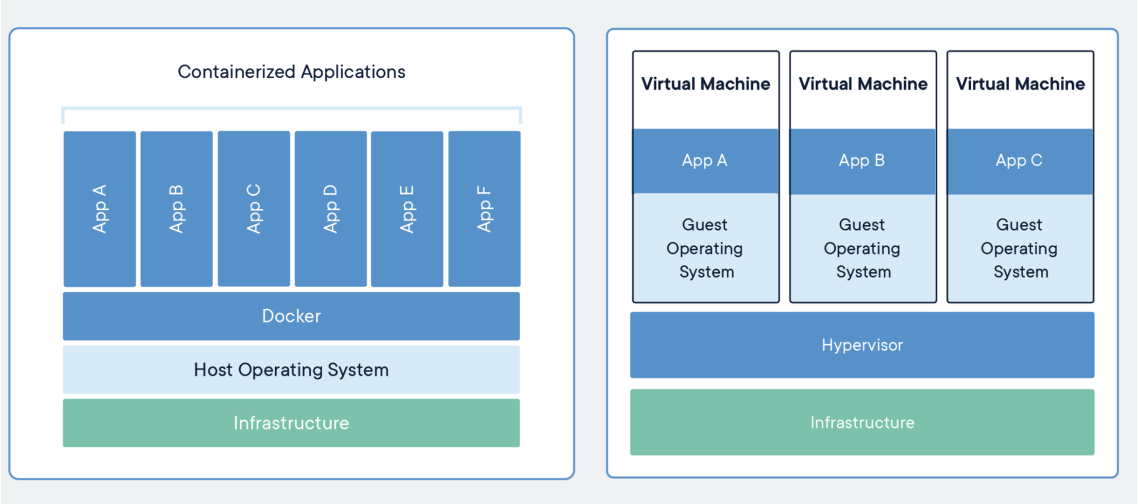

Figure 14: Figure explaining the difference between Containers and Virtual Machines
Source:[8]

Docker [9] is the most widely accepted utility to create containers and is used in ttnV2 deployments, which will be retained in the ttnV3 deployment process.

4.1.2 Step 2: Installation

In order to run the software, Virtual Machines have to be prepared with the appropriate OS image, supporting packages and configured with sufficient memory. There are two currently possible targets for deployments based on the type of resource used:

- Cloud VMs
  In this case, the ttnv2 runs on compute resources provided by a commercial cloud provider. Most of the Public networks run on Microsoft Azure while in private networks, AWS is preferred by clients. The VM instances are managed either by TTN or by the client themselves.
- On-Premise
  Here, the stack runs on a private server where the customer provides access to. In this case, the servers are managed by client themselves.

The installation of the software is a multi-stage process involving:

- Installation of the OS and support packages with appropriate permissions.
- Installation of Secret Keys and certificates for Secure discovery and communication.
- Initiation of supporting services such as Databases and Message Brokers.
- Initiation of individual micro-services.
- Basic tests for connectivity and functionality.

For ttnv2, installation and configuration *performed manually for each installation* using handwritten scripts. This process is tedious and requires manual supervision. On average, *an hour's worth* of effort is necessary to configure a new network and this method is not scalable when there are hundreds of customers.

4.1.3 Step 3: Post-Installation (Operations)

Software Operations (Ops) is the process of monitoring and maintaining a software installation in production. Due to the distributed nature of ttnV2 deployments, the operations become complex and tedious. A few important operational considerations are outlined here.

- **Monitoring and Logging:**
  An essential component of any system in production the ability to monitor various run-time parameters and store these logs for debugging purposes. Monitoring and logging are separate operations themselves requiring a multitude of components as shown in the image below:

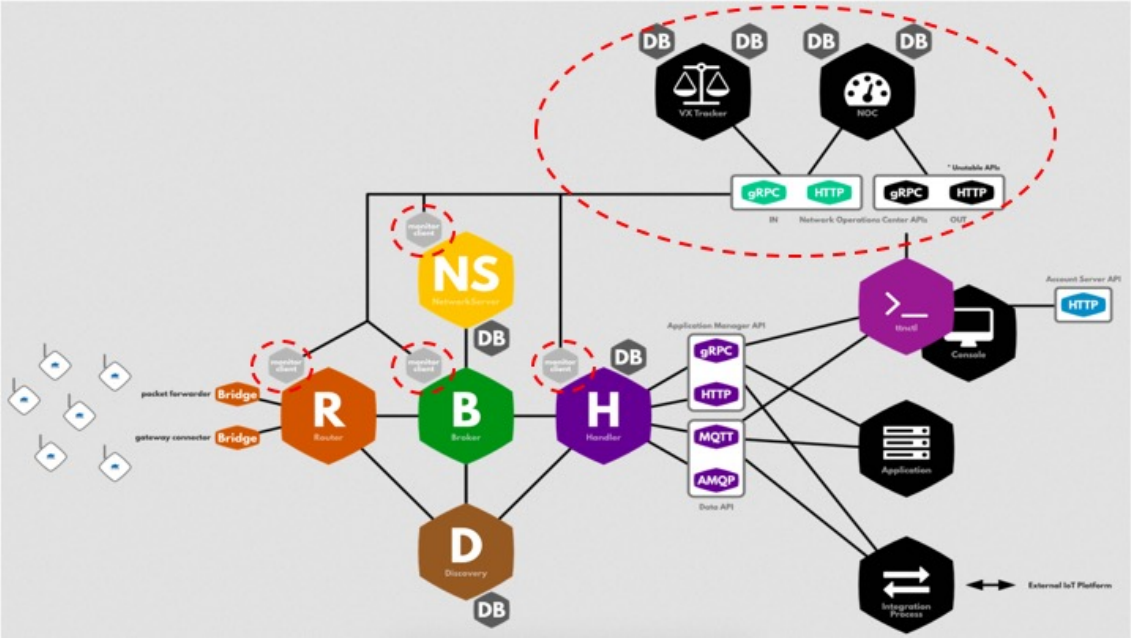

Figure 15: Enhanced View of a ttnV2 installation with Logging support
Source:[7]

The above diagram represents the key type of metrics namely gauges (scales) which can increment or decrement and counters (meter) which can only increment. Due to diverse deployment scenarios, each instance needs to be monitored separately, which cannot be scaled apart from scaling up the man-power.

- **Fault Diagnosis and Recovery:**
  In the event of system crashes or *alerts* as reported by the monitoring mechanisms, it is essential to locate the cause of the issue and provide run-time fixes. The current method is to enter into a remote shell (using ssh) on the target VM, check the logs and restart or update the particular service/s that is/are failing. This technique is tedious and requires manual intervention by Ops engineers, which *will not scale* as the number of clients increase.

- **Rolling updates and upgrades:**
  There are a few instances where the services running on VMs need updates, such as bug fixes, license updates and memory expansion to accommodate an increase in traffic. Unfortunately, in the current process, these updates have to be performed manually by Ops engineers which is not scalable.

4.1.4 Problems

As we have analyzed the ttnV2 deployment and operation processes, the following problems were discovered:

- Each instance of the software stack needs to be installed (deployed) and configured on the target machine separately, which scales poorly as more tenants are added.
- Each instance has to be monitored independently for proper functioning. This process is tedious as it requires dedicated manpower which, once again, does not scale.
- When bugs/issues are found that affect the common software for all the instances, bug-fixes need to be applied individually to each instance, which then needs to be monitored to make sure that the fixes are properly working.

This lack of scalability of single-tenant distributed systems is the motivation to explore a different deployment scheme/s for the new ttnV3 software stack. This chapter is dedicated to exploring and defining a new deployment scheme based on the SaaS (Software As A Service) paradigm.

4.2 Research Questions

The following questions are derived from the shortcomings of the existing process explained in the previous section. The rest of this chapter is dedicated to finding answers to these questions.

- What architectural patterns exist for SaaS deployments?
- How are micro-services coordinated in a SaaS architecture?
- What are the chief concerns that need to be addressed while designing a SaaS system with micro-services?
- Which are the possible deployment schemes for ttnV3 stack based on its concerns?
- What tools/frameworks are available to deploy and manage these micro-services?
- How can the various concerns be tested and what tools exist to do so?

4.3 Inputs from Literature Study

In order to design a new deployment scheme based on the Software-As-a-Service(SaaS) principle, it is essential to understand it in the first place, to which the rest of this section is dedicated to. SaaS is defined in [10] as "*a form of Cloud computing in which applications are hosted by a service provider and made availability to customers over a network, typically the Internet*".

4.3.1 Related Work

There is an ocean of literature available on SaaS architectures. A very good overview of the architectural concerns in SaaS systems is presented in [11] and in [12]. [13] defines and discusses models that help in achieving configurability is SaaS systems. The authors of [14] present a thorough framework for application customization, whereas [15] presents an algorithmic approach for the placement of tenants based on their resource utilization. [16] presents an overview of CPU utilization in a multi-tenant cloud whereas [17] provides useful insights into SaaS architectures based on micro-services. And finally, [18] and [19] provide mathematical results on handling databases in multi-tenant clouds.

4.3.2 Classification of SaaS Architectures

With the aforementioned literature serving as a reference, this paper identifies three major SaaS architectural patterns. Before discussing these patterns, key concepts such as *tenants* and *multi-tenancy* need to be defined. Tenants are merely customers who use the SaaS service(software) and Multi-tenancy is defined in [12] as "*an architectural pattern in which a single instance of the software is run on the service provider's infrastructure, and multiple tenants access the same instance*". The software in this context refers to a service, in this case the overall routing service provided by the TTN Stack. With this context set, the three major SaaS architectural patterns are described below:

1. Single Tenant Single Instance(STSI):
   This is the simplest form of a SaaS system where there is only one tenant connected to a SaaS service, which is hosted by the software provider. A single server in ttnV2 deployment can be considered as a single tenant centralized deployment. This scheme is mentioned here only for the sake of completeness and will not be further explored in this text.

2. Multi Tenant Single Instance(MTSI):
   In this scheme, a single instance of the software stack serves multiple tenants. Each tenant connects to the same server and it is the responsibility of the server to be able to handle the load of the requests from the tenants without sacrificing the performance to any particular tenant. In reality, systems use proxies as a front end which route requests to servers that may themselves be distributed in nature.

3. Multi Tenant Multi Instance(MTMI):
   In this scheme, several multi-tenant instances are deployed in a distributed manner and any particular tenant may choose any one of these servers based on factors such as geographical proximity or shortest response time. An example of the former are regionally distributed services such as regional Amazon Web Service consoles and as for the latter, a good example is the Peering Broker using a discovery service to find the best server to serve a proxy.

4.3.3 SaaS and Containers

At this juncture, it is crucial to set the scope of the discussion for this chapter. Since the services are packaged into containers which are then installed onto machines (real or virtual), the focus of this chapter is on how these containers can be properly managed. The authors of [20] provide a good example of how containers can exist on top of SaaS infrastructure, though they stray into implementation details in their analysis. Fig 16. is abstracted from their analysis at a higher architectural level.

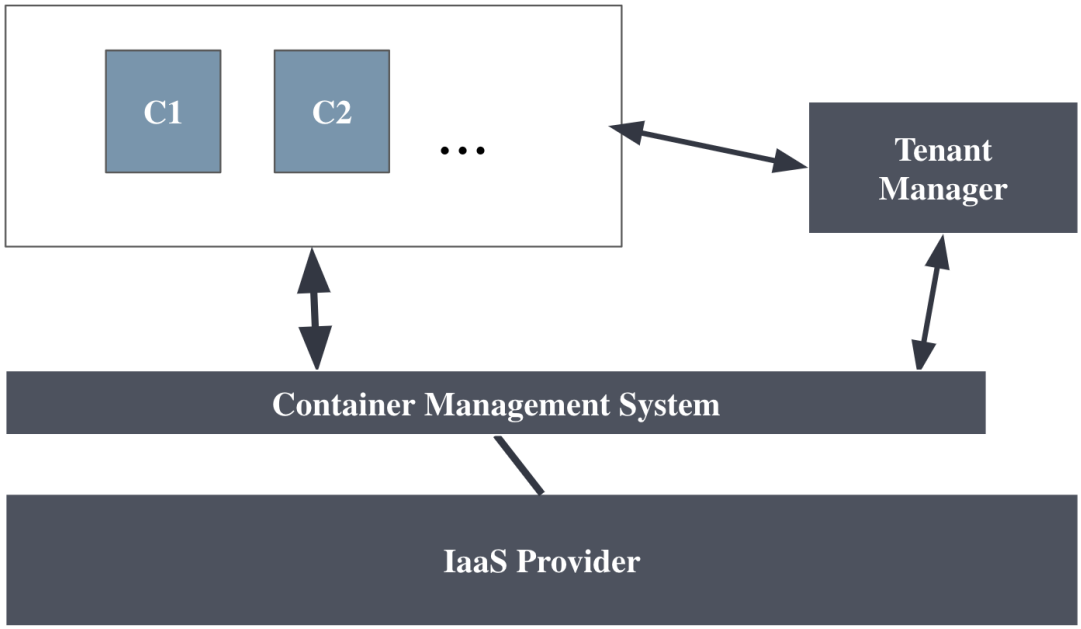

Figure 16: Containers over an IaaS Provider

4.3.4 Important definitions

- Clustering:
  This paper defines clustering as the process of logically grouping containers and defining the communication methods between them.

- Cluster Management:
  Cluster Management is concerned with handling with the clusters of containers as defined above.

4.4 Design

Using the inputs from the literature, we can now explore a design for the SaaS deployment scheme for the ttnV3 stack. In order to do so, we must first understand how the microservices detailed in Chapter 3 are mapped into Docker containers, which factors (concerns) influence this mapping and which amongst them have the highest priorities.

4.4.1 Mapping of ttnV3 services to containers

In terms of Architectural Descriptions, software packages are referred to as *components* and the resulting diagram is called a *component diagram*. The following image shows the components(micro-services) for ttnV3 systems and their interaction. This figure builds up on the architectural description made in Chapter 3 by treating each service as a component. End Device and Gateways are physical entities whereas the peering broker (as described in chapter 4) is made up of multiple components itself. Therefore these entities are not considered as components. As for the micro-services that are actually considered as components in the diagram(ex: JS or NS), they are directly packaged into a docker containers, one for each micro-service. This packaging is done by defining a configuration file (*.yaml* file) and invoking the Docker utility with this file as an input. A sample configuration file along with the commands to invoke the docker utility are available in the appendix. It must be noted that this process is exactly the same as for ttnV2 micro-services.

4.4.2 Architectural Concerns for ttnV3 SaaS

Based on the literature research and technical discussions with the cloud architects at TTI, there are two types of concerns that affect the architectural decisions, namely Properties and qualities. The following tables lists some important properties and qualities of a SaaS system along with the desired states.

Table 13: Properties of a SaaS system and the desired states

Property	Description	Desired State
Affinity	Defined as the binding of Users(tenants) to particular instances.	Affinity should be low and dynamic for better resource utilization.
Performance Isolation	Defined as the degree of impact on a tenant due to over-usage by another tenant.	Should be kept to a minimum.
Customization	Defined as the degree to which a tenant can tailor the performance of the system to a certain degree to suit its needs	Customization depends usually on the specific use-case.
Resource Sharing	Defined as the degree to which resources such as OS/Memory are shared between tenants.	Must optimally balance resource consumption and complexity in resource sharing schemes.

The following table lists the various qualities of a SaaS system and their desired levels.

Quality	Description	Desired Level
Updatability	Defined as the ability of the system to easily rollout software updates without affecting performance.	It's more beneficial to have different staging and production environments to isolate the testing of newer versions.
Scalability	Defined as the ability of a system and supporting tools to seamlessly allow the extension of the services to new tenant.	The average cost per tenant must remain constant (with some tolerance) with elasticity of tenants.
Observability	Defined as the ability of the system to provide a monitor with fine-grained metrics and logs that can be used to improve performance and detect issues.	There are various levels of information which will be discussed later in this chapter.
Meterability	Defined as the ability of the system to store fine-grained usage data per-tenant for the purposes of billing.	Since ttnV3 currently employs a per-device billing scheme, there only needs to be metrics on devices per-tenant and that can easily achieved by querying a database.

4.4.3 Proposed Schemes

Considering the requirements above, this paper proposes three clustering schemes on the SaaS infrastructure. In table 14, the different clustering schemes proposed above are evaluated against the requirements and provided a *label* that is relative to each other. Most requirements are assigned "*high, intermediate or low*" relative to each other and others are assigned "*easy, intermediate or difficult*", again relative to each other.

Table 14: Evaluation of proposed Clustering schemes against the requirements

Requirement	Tenant-Affine	Non-Affine	Split-Affine
Affinity	High	Low	Intermediate
Performance Isolation	High	None	Low
Customization	High	None	Intermediate
Resource Sharing	Low	High	Intermediate
Software Updates	Easy	Easy	Easy
Scalability	difficult	easy	intermediate
Observability	easy	difficult	intermediate
Metering	easy	difficult	intermediate

4.5 Practical Considerations

Up to this point in the document, we concerned ourselves only with theoretical analysis. In this section, we look at some practical considerations which includes evaluation potential tools for creating, managing and updating container clusters.

4.5.1 Overview of Tools

The following table lists the tools chosen for different stages and the rationale for the choice.

Table 15: Tools for the various stages of deployment

Purpose	Tool/Service	Rationale
Container Creation	Docker	Docker is widely accepted and is almost synonymous with containers.
Container Storage	Gitlab Registry	Gitlab registry is free to use and is a private repository.
Cluster Deployment and Management	Kubernetes	Kubernetes is the current front-runner in container management.
Rolling Updates	Spinnaker	Spinnaker was built to integrate well with Kubernetes.
Monitoring/Metrics	Prometheus	Prometheus is a well known open-source highly configurable metrics aggregator.
Visualization	Grafana	Grafana is highly configurable dashboard with various templates and works well with prometheus.

4.6 Results

As a result of the analysis in this chapter, we have the following

- A clear understanding multi-tenancy on SaaS, its properties and qualities.
- Three SaaS deployment options to choose from based on tenancy.
- Various tools to assist in the deployment lifecycle have been identified and evaluated (details in the appendix).

4.7 Summary

In this chapter we examined the ttnV2 deployment process, identified its issues. We explored the architectural concerns of ttnV3, including the study of various properties and qualities of a

multi-tenant SaaS system. Using this analysis, three schemes were proposed based on the type of affinity. Finally, we explored various tools that would facilitate the deployment process. The next chapter explores mechanisms to exchange traffic between networks.

5 Collaboration between networks

The Things Network is in fact a collection of multiple network instances, each serving a specific purpose. However, as described in the introductory chapter, since the instances serving individual customers are isolated from one another (and from the Public Community Network) and since there is no mechanism to exchange traffic between networks, the notion of one global network does not yet exist. In this chapter we discuss the design and implementation of a system called the Peering Broker (peering is the TTN terminology for traffic exchange between networks) to enable collaboration between networks.

5.1 Problem Context

The Things Network(and LoRaWAN in general) is an open network aiming for global coverage and connectivity. The success of the network is directly proportional to the number of participants and the cooperation between them. However, neither the ttnV2 stack nor the supporting LoRaWAN 1.0 specifications provided any mechanism for Networks to exchange data. This resulted in a fragmented system where each network served only a dedicated number of devices. This isolation is depicted in the figure below:

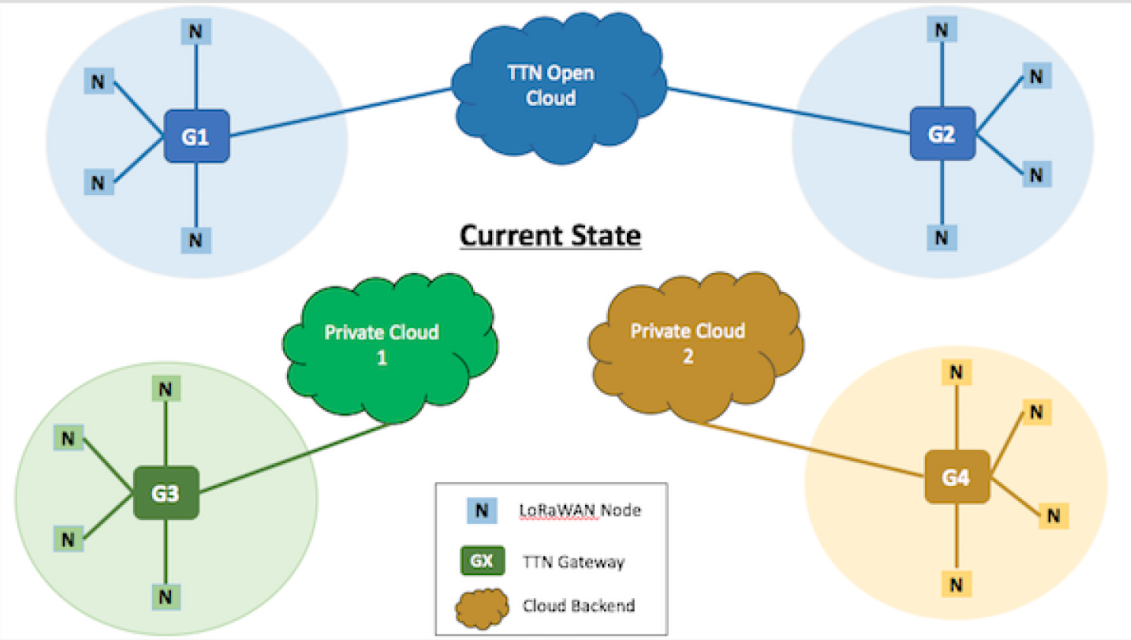

Figure 17: ttnv2 Deployment scenarios

Consequently, devices can operate only if they are present in a location covered by the network (home network) which handles those devices. But field data suggests that regions of (private/public) networks typically overlap. This restricts the operation of the device only to certain regions and the networks themselves have redundant overlap in certain geographical area. The LoRa Alliance has addressed this short-coming in the latest version of the specification by introducing the concept of *Roaming*. The rest of this chapter will examine these specifications and their consequences in the ttnV3 design in detail. Before doing so however, it is essential to define the term *Network Server*.

5.1.1 Disambiguation of the Network Server

The LoRaWAN Specifications refer to the Network (Backend) as the *Network Server*. As explained in the previous section, the ttnv3 stack refers to the component (service) that handles the network layer of a device as the ”Network Server”. For the sake of disambiguation, in this document, the term ”Network Server” is reserved for the ttnv3 component. The *LoRaWAN Network Server* will be referred to plainly as the ”Network”.

5.2 Requirements

The Requirements for the TTN Peering Infrastructure are as follows.

1. Functional:
   (a) Support LoRaWAN 1.1 Roaming functionality (explained in detail later in this section).
   (b) Implement a mechanism to calculate the number of packets transmitted between networks.
   (c) The traffic that is exchanged must be encrypted and integral.
2. Non-functional:
   (a) The system must be easily scalable i.e., it must be easy to add/remove networks without affecting performance.
   (b) The system must be tolerant to faults.
   (c) The system must be observable, i.e., it must be easy to monitor and log events during run-time.
3. The Roaming functions maybe optionally extended to include LoRaWAN 1.0 devices as well.

5.3 LoRaWAN Literature

In this section, we discuss the LoRaWAN Backend Interfaces (1.0) [21] to understand the prescribed specifications on Roaming of Devices, which in-turn is facilitated by Peering between consenting networks.

5.3.1 Roaming

The LoRaWAN Specification defines three states for a Network Back-end (refered to as Network Server/NS in the specification) with regards to handling a particular device:

1. Home NS (hNS):
   This the NS which the device and the application is registered to. This is the *final destination* of Device uplink messages. This NS also is connected to the Join Server where the device’s keys are held. When a device is connected to this NS directly, there is no notion of Roaming involved.
2. Forwarding NS (fNS):
   In this state, the NS simply forwards uplink/downlink packets to the Home NS. It does not maintain any state nor does it control the device’s activity.
3. Serving NS (sNS):
   In this state, the NS handles the complete MAC layer operations of the device including maintenance of Device state, control of device operations, encryption and decryption of Network commands and Integrity checks.

5.3.2 Types of Roaming

The LoRaWAN Backend Interfaces 1.0[21] defines two types of roaming based on the state of the NS to which the device is connected:

1. Passive Roaming:
   In this case, the NS connected to the device acts only as a Packet Forwarder (fNS). The device state and control information is held by another NS (sNS/hNS)

2. Handover Roaming:
   In this case, the NS (sNS) connected to the device takes over the Network control of the device and only the application packets are routed to the home NS.

The Fig 22 is borrowed from the LoRaWAN spec:

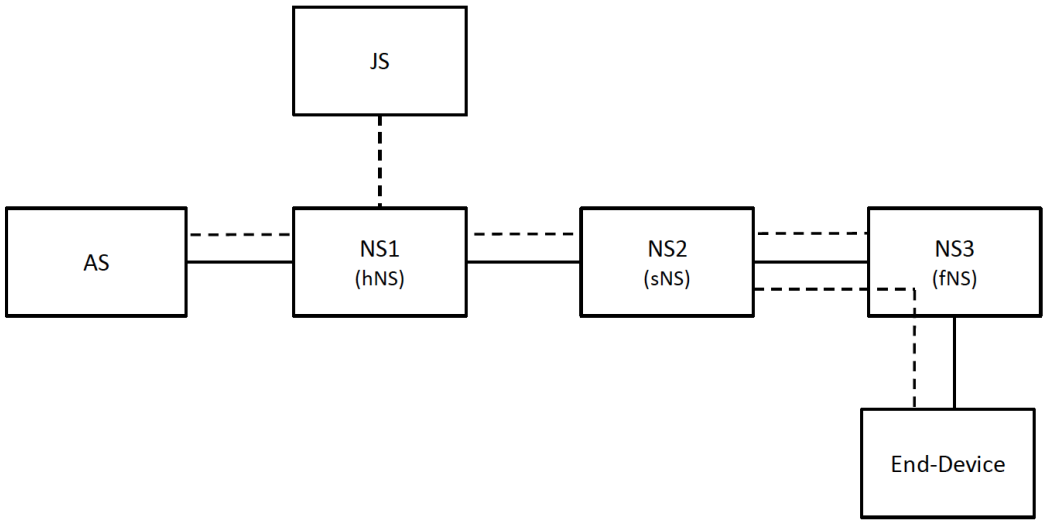

Figure 18: Network server states during Peering as defined LoRaWAN Backend Interfaces
Source:[21]

5.3.3 Device Activation while Roaming

When a device moves from one region to another, it must send a Join request to create a new session i.e., *every kind of Roaming must start with a Join request*. This is also a trigger the network servers involved to enter into a roaming arrangement with each other to best serve the device.

5.3.4 Session State while Roaming

The section on LoRaWAN explained the fact that every Join/Rejoin Request created a new session for the device and this is stored in the Network Server. During passive roaming, the session of the device is maintained in the Home Network since the forwarding Network does not maintain device state. However, in the case of the handover roaming, the serving Network, maintains the state of the device, which gets reset upon every Join/Rejoin request.

5.3.5 LoRaWAN Reference Architecture

The LoRaWAN Backend Interfaces(v1.0) [21] defines the following *standard* implementation of Networks during Roaming:

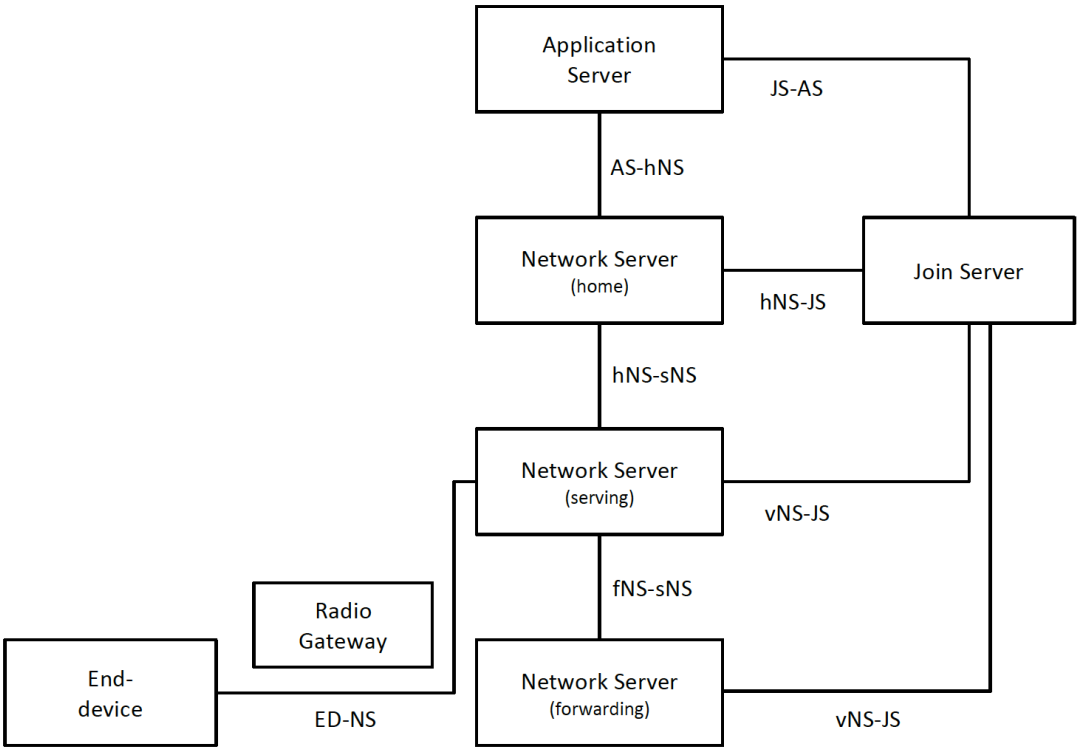

Figure 19: LoRaWAN roaming reference implementation archtitecture
Source: LoRaWAN Backend Interfaces(v1.0)[21]

5.3.6 Issues with reference Architecture

The proposed LoRaWAN roaming reference implementation archtitecuture has the following drawbacks.

- Common Join Server (JS):
  In order for devices to join the serving network backend, the devices have to send a Join Request which has to be routed to the correct Join Server (JS) to validate the device. The Specifications propose that all the Network Backends that support roaming be connected to a particular JS that holds the keys to that particular device. This makes the JS a Single Point of Failure (SPOF). Furthermore if this JS is compromised, then all the networks that use keys from it are vulnerable.

- 1:1 connections:
  The reference implementation proposes that the Network backends that enter into a roaming arrangement create 1:1 data streams for sharing data. LoRaWAN packets are designed to be short and periodic and it is a waste of resources to maintain persistent connections for such data. Also, these connections are not scalable as the number of backends increase.

5.4 Design

The fact that common Join Servers are a threat to the security of the LoRaWAN networks and that maintaining persistent connections between networks is not optimal for LoRaWAN networks encouraged us to rethink the design for our peering system.

5.4.1 PubSub Architectural Pattern

The Publish-Subscribe (PubSub) as described in [22] is an Event-Based coordination pattern which is best suited for systems where there is a notion of entities that create data (Producers/Publishers) and entities that use this data (Consumers/Subscribers) and where there is a need to decouple the two entities. Since this pattern closely fits the requirements for the ttnV3 peering system, we introduce a new entity called the Peering Broker (PB), which works in accordance with the PubSub pattern.

5.4.2 Architecture

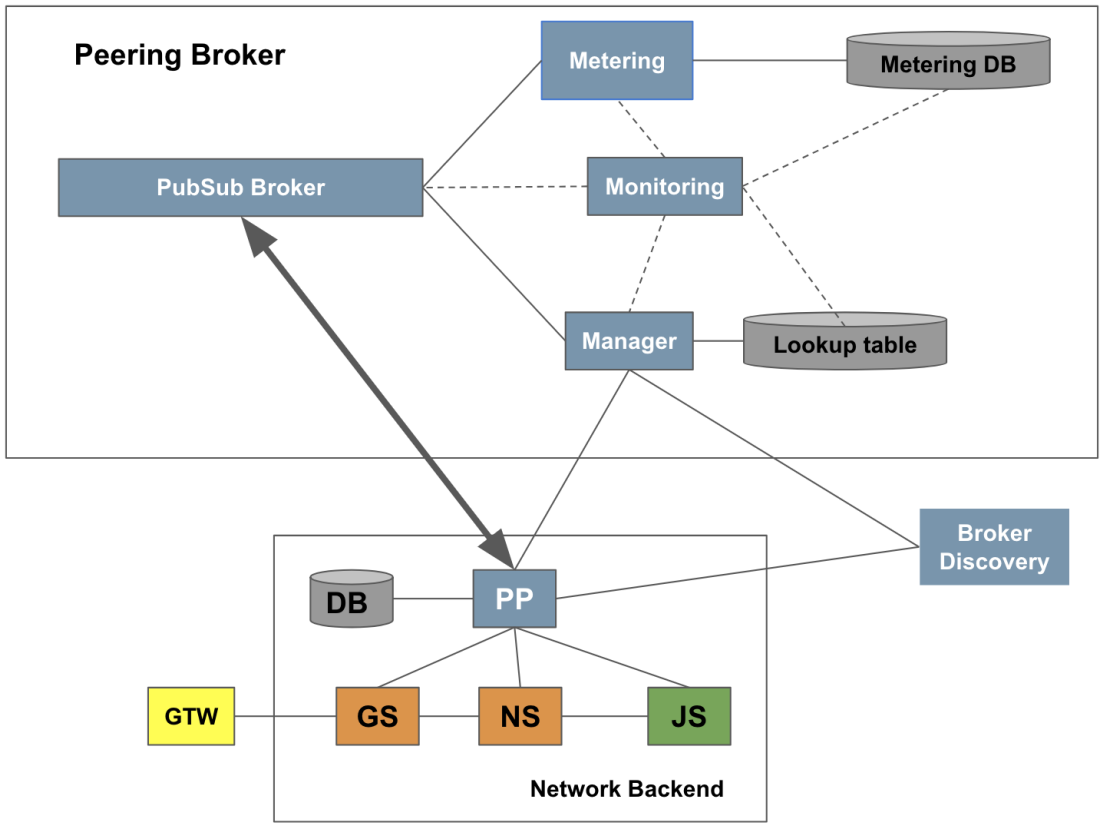

Figure 20: TTN V3 Peering Architecture

The Peering Broker is itself composed of multiple MicroServices, the most vital amongst them being the PubSub Broker. This is the component that mediates between the Publishers and subscribers. The Publishing Network *publishes* data to the this PubSub Broker and other Networks that are interested in this data *subscribe* to this data. The data is bifurcated based on a *Topic*, which serves as the reference for publishing and subscribing data. The other microservices of the PB are constructed to support the PubSub broker. The main functions of each micro-service is described in table 16.

Table 16: Overview of the Components in the Peering Broker

Component	Main Function
PubSub Broker	A message broker that works on the Publish-Subscribe principle.
Manager	Manages the connection of networks, stores and retrieves keys.
Monitoring	Retrieves application and network metrics from various components.
Metering	Keeps track of the number of messages passed between networks.
(Broker) Discovery	Acts as a fixed end-point which networks can query for the location of the Broker making the broker instances dynamic. This service can either be hosted on the same machine as the other services or separately.
Look-Up	Used to store and retrieve network information.
Peering Proxy	An component that is part of the ttnV3 stack that interacts with the Peering Broker and the other components in the stack, thus enabling inter-network peering.

5.5 Implementation Details

5.5.1 Api Definitions

The micro-services design enforces loose coupling between services by the use of APIs. We prefer to use Protobuf[23] to define the interface between the services. An example of a proto api is shown in Fig 30

```
message Uplink{
    // phy_payload is the complete LoRa physical payload.
    bytes phy_payload = 1 [(gogoproto.nullable) = false];
    // mac_payload is the mac layer payload.
    bytes mac_payload = 2 [(gogoproto.nullable) = false];
    // app_payload is the application payload. This field is used only during Handover Roaming.
    bytes app_payload = 3 [(gogoproto.nullable) = false];
    // dev_addr is the 24 bit Address of the device that sent this uplink.
    bytes dev_addr = 4 [(gogoproto.nullable) = false];
    // ul_metadata is the device metadata associated with this uplink.
    bytes ul_metadata = 5 [(gogoproto.nullable) = false];
    // gtw_tx_setttings are the settings used by the device to send this uplink.
    bytes dev_tx_setttings = 6 [(gogoproto.nullable) = false];
    // signed_hash contains the hash of the rest of the message, signed by the network's private key.
    bytes signed_hash = 7 [(gogoproto.nullable) = false];
}
```

Figure 21: Protobuf definition of an uplink message

This interface is then used in a service call using gRPC (google Remote Procedural Call) [24]. The protobuf compiler can automatically create the underlying code in any of the supported languages (golang being one of them). This generated code is then used in the service to call the RPC and pass the data as defined in the message.

5.5.2 Metering

All publishes/subscribes are logged by the Broker using the Metering component. This data is stored in a separate database and can be queried for the quantity of inter-network traffic, which can be used for billing between the networks.

5.6 Additional Considerations

5.6.1 Data Security

The system uses a combination of asymmetric and symmetric encryption schemes in order to allow flexibility and also to ensure data security.

5.6.2 Integrity Check

Standard cryptographic signature schemes are used to ensure integrity of messages exchanged between the networks.

5.6.3 Isolation

Since the PubSub does not explicitly restrict networks from subscribing from Topics, Networks can technically subscribe to packets intended for other networks. Though the packets are encrypted, the time of arrival of packets and their length can be used to perform analytics on the peering networks which can reveal some interesting information. This is not handled in this implementation.

5.7 Validation

5.7.1 Test Setup

The Peering Broker and the Proxy are tested using a tester, which is a software component that simulates the rest of the stack. The Peering Proxy API is called with test inputs its results are logged in Docker as well as on the dashboard.

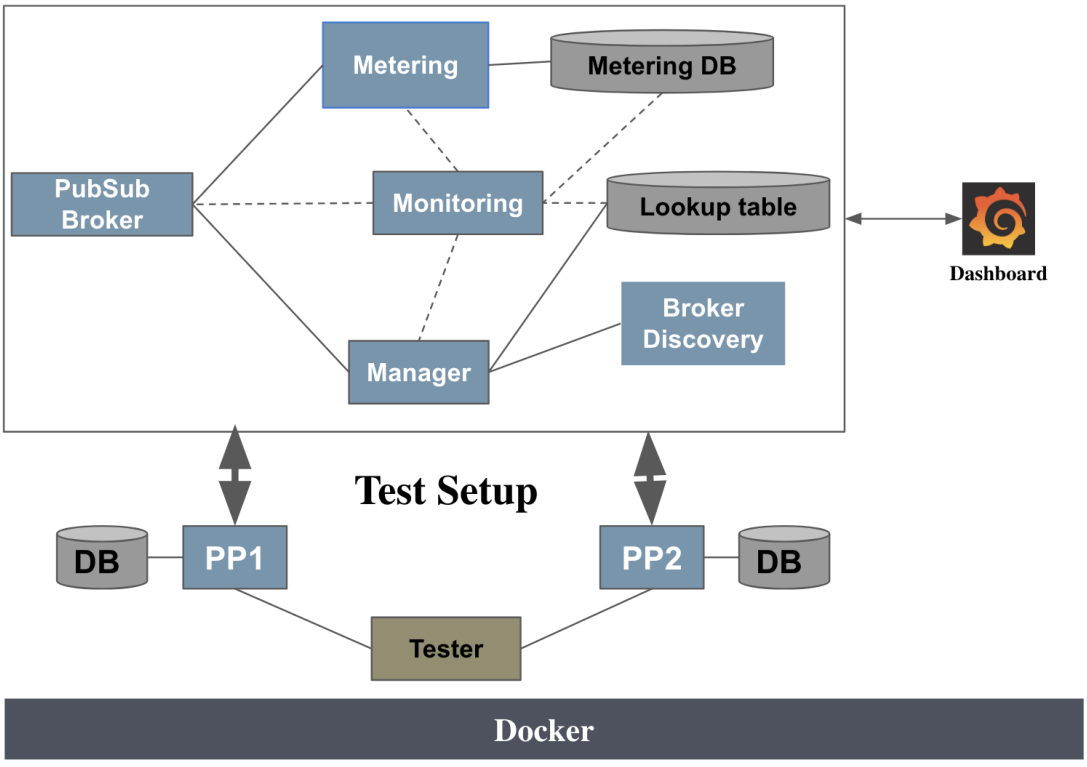

Figure 22: Peering Test Setup

5.7.2 Sample Results

The following figure shows the final results of a simulation with two networks with randomized inputs as reported by the dashboard. For a more detailed explanation of this simulation, please refer to the appendix.

Figure 23: Sample Test Result as seen on the Dashboard

5.8 Summary

In this chapter we examined the TTN Peering Broker, a system designed to enable the exchange of traffic between networks. Its PubSub based design addresses the issues with the LoRaWAN standard reference implementation. Networks can exchange traffic without needing to connect to a single Join Server and without maintaining persistent 1:1 connections. This improves the scalability, load-handling capability and fault-tolerance of the system as demonstrated by the tests. The Peering Broker also maintains peering metrics that can be used by networks for billing purposes and the use of a symmetric encryption scheme aided by the asymmetric security scheme meant that the networks can exchange data securely.

6 Conclusions

This project was executed with the goal of improving the scalability and connectivity between various instances of The Things Network software stack, while leveraging the upcoming version three (ttnV3) of the architecture. In order to attain this goal, it was essential to, at the outset, understand ttnV3 from an architectural point of view. We did this by first investigating the existing stack (ttnV2) and its shortcomings to sketch the motivation for ttnV3. Then we modeled ttnV3 structurally and behaviorally to scrutinize its design and compared it with a monolithic LoRaWAN network implementation (LoRaServer), as well as ttnV2 to appreciate the benefits offered by its architecture. The resulting models and the analysis documented here serve as a reference to understand ttnV3.

Upon gaining a clear understanding of the architecture of ttnV3, the next step was to analyze the current deployment process used for ttnV2, address its scalability issues, and propose solutions to improve this process for ttnV3. We did so by first dissecting the ttnV2 deployment process into three stages and diagnosing the problems in each stage. This led to the conclusion that the *single tenant, multi-instance* approach currently used was not conducive to scalability, and hence, a new approach was needed. Consequently, we explored the Software As A Service (SaaS) paradigm for potential solutions. Since ttnV3 uses the micro-service design paradigm, additional considerations needed to be made to garner the benefits of SaaS. We then elucidated three SaaS architectures, their qualities and properties, which led to the proposal of three clustering schemes. Recognizing the fact that advanced tool support is necessary to realize SaaS with microservices, we explored state of the art tools to support our deployment including Kubernetes, the current leading solution in container management. Though the ttnV3 stack is not yet available, this research has laid the groundwork necessary for easing the deployment process when the stack is available.

The next important phase in making TTN scalable is to introduce cooperation between the network instances. The LoRa Alliance paved the way for this feature by introducing the concept of Roaming in the latest specifications (v1.1). But, the analysis of these specs soon revealed architectural issues that would inhibit scalability in networks implementing these specifications. As a result, we redesigned the peering architecture using the Publish-Subscribe paradigm by introducing a service called the *Peering Broker*. The various stages of designing and implementing this system were documented here. This system was validated by creating a simulation network to provide test inputs to the Peering Broker and its proxy and the results were visualized on a dashboard. These results demonstrated how the pubsub mechanism is able to handle high loads, how the peering broker is able to seamlessly add or remove networks, and how the system as a whole can be monitored using detailed logging mechanisms.

The final ingredient in making ttnV3 truly scalable is the quantification of the messages passed between networks in order to estimate the volume of traffic exchanged between them. This was easily accomplished by the metering component in the broker which subscribes to all the data that is passed between the networks and keeps a detailed log of the messages transferred.

6.1 Future Work

The information presented and the results demonstrated in this project serve as the basis for the future work which is summarized below.

- SaaS Deployment
  As of writing this document, the team at TTI is working hard on creating a Minimum Viable Product (MVP) of the ttnV3 stack, which includes the most basic functions needed by device and gateways. Once this MVP is available, the data presented in this document can be used to implement different clustering mechanisms to obtain statistics to validate the theoretical arguments and to use the tools analyzed here with these clusters.

- Peering in ttnV3
  The Peering Broker (and the peering proxy) implemented in this project is currently tested in isolation with test inputs for its functioning. However, once the MVP is available this system can be integrated with the ttnV3 stack to test with real inputs. Subsequently, both ttnV3 and the peering broker can be hosted on a suitable cloud to ease their maintenance.

- Compensation
  Finally, the metering information that is collected and stored by the metering component can be extended with other metrics, such as gateway usage information, in order to compensate the gateway owners for their contribution to the network. This would involve placing such statistics on an open platform such as a *blockchain* and using mechanisms such as *smart contracts* to automate the compensation process.

References

[1] W. Giezeman, "The things network: Building a global iot data network in 6 months," Jan 2016. [Online]. Available: https://medium.com/@wienke/the-things-network-building-a-global-iot-data-network-in-6-months-adc2c0b1ae9b

[2] TheThingsNetwork, "Thethingsnetwork/manifest." [Online]. Available: https://github.com/TheThingsNetwork/Manifest

[3] "What is lora®?" [Online]. Available: https://www.semtech.com/technology/lora/what-is-lora

[4] "Lorawan™ regional parameters v1.1rb — lora alliance™." [Online]. Available: https://lora-alliance.org/resource-hub/lorawantm-regional-parameters-v11rb

[5] "Lorawan™ specification v1.1 — lora alliance™." [Online]. Available: https://lora-alliance.org/resource-hub/lorawantm-specification-v11

[6] "Microservices," May 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[7] "Network architecture," Aug 2018. [Online]. Available: https://www.thethingsnetwork.org/docs/network/architecture.html

[8] "Playing catch-up with docker and containers," Feb 2017. [Online]. Available: https://rancher.com/playing-catch-docker-containers/

[9] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[10] H. Kriouile, Z. Mcharfi, and B. E. Asri, "Towards a high configurable saas - to deploy and bind auser-aware tenancy of the saas," in *Proceedings of the 17th International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, INSTICC. SciTePress, 2015, pp. 674–679.

[11] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant saas applications." in *CLOSER*, F. Leymann, I. I. Ivanov, M. van Sinderen, and T. Shan, Eds. SciTePress, 2012, pp. 426–431. [Online]. Available: http://dblp.uni-trier.de/db/conf/closer/closer2012.html#KrebsMK12

[12] C.-P. Bezemer and A. Zaidman, "Multi-tenant saas applications," *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE) on - IWPSE-EVOL 10*, 2010.

[13] H. Wang and Z. Zheng, "Software architecture driven configurability of multi-tenant saas application." in *WISM*, ser. Lecture Notes in Computer Science, F. L. Wang, Z. Gong, X. Luo, and J. Lei, Eds., vol. 6318. Springer, 2010, pp. 418–424. [Online]. Available: http://dblp.uni-trier.de/db/conf/wism/wism2010.html#WangZ10

[14] W.-T. Tsai and X. Sun, "Saas multi-tenant application customization." in *SOSE*. IEEE Computer Society, 2013, pp. 1–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/sose/sose2013.html#TsaiS13

[15] E. Yang, Y. Zhang, L. Wu, Y. Liu, and S. Liu, "A hybrid approach to placement of tenants for service-based multi-tenant saas application." in *APSCC*, J. J. Park, C. Nikolaou, and J. Cao, Eds. IEEE Computer Society, 2011, pp. 124–130. [Online]. Available: http://dblp.uni-trier.de/db/conf/apscc/apscc2011.html#YangZWLL11

[16] G. Velkoski, M. Simjanoska, S. Ristov, and M. Gusev, "Cpu utilization in a multitenant cloud." in *EUROCON*. IEEE, 2013, pp. 242–249. [Online]. Available: http://dblp.uni-trier.de/db/conf/eurocon/eurocon2013.html#VelkoskiSRG13

[17] S. Kalra and T. V. Prabhakar, "Towards dynamic tenant management for microservice based multi-tenant saas applications." in *ISEC*, Y. R. Reddy, V. Varma, J. Cleland-Huang, U. Bellur, S. Sengupta, N. Sharma, R. Loganathan, R. Sharma, and S. Sarkar, Eds. ACM, 2018, pp. 12:1–12:5. [Online]. Available: http://dblp.uni-trier.de/db/conf/indiaSE/isec2018.html#KalraP18

[18] E. A. Boitsov, "Load balancing and data-management strategies in a multitenant database cluster." *Automatic Control and Computer Sciences*, vol. 48, no. 5, pp. 282–289, 2014. [Online]. Available: http://dblp.uni-trier.de/db/journals/accs/accs48.html#Boitsov14

[19] A. R. Ortega, M. Noguera, J. L. Garrido, K. B. Akhlaki, and J. Barjis, "Extending multi-tenant architectures: a database model for a multi-target support in saas applications." *Enterprise IS*, vol. 10, no. 4, pp. 400–421, 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/eis/eis10.html#OrtegaNGAB16

[20] E. Truyen, D. V. Landuyt, V. Reniers, A. Rafique, B. Lagaisse, and W. Joosen, "Towards a container-based architecture for multi-tenant saas applications." in *ARM@Middleware*. ACM, 2016, pp. 6:1–6:6. [Online]. Available: http://dblp.uni-trier.de/db/conf/middleware/arm2016.html#TruyenLRRLJ16

[21] "Lorawan™ back-end interfaces v1.0 — lora alliance™." [Online]. Available: https://www.lora-alliance.org/resource-hub/lorawantm-back-end-interfaces-v10

[22] A. S. T. Maarten van Steen, *Distributed Systems. 3rd edition, v3.01*, 2017.

[23] [Online]. Available: https://developers.google.com/protocol-buffers/

[24] "grpc open-source universal rpc framework." [Online]. Available: https://grpc.io/docs/guides/

Appendix

Appendix 1: Container Management Platforms

Most container management solutions can be placed in two broad categories.

- Tool-based:
  They are usually open-source software which need to be manually integrated into/packaged with the ecosystem of systems under question and manually managed by the software team. The advantage with using tools is their flexibility and the ability to customize them for the use case.

- Service-based:
  Services are usually closed-source, fully integrated solutions that are built using the *tools* provided to the software team as a service. The advantage of this approach is that there is no need to maintain the container management software in addition to the rest of the software components.

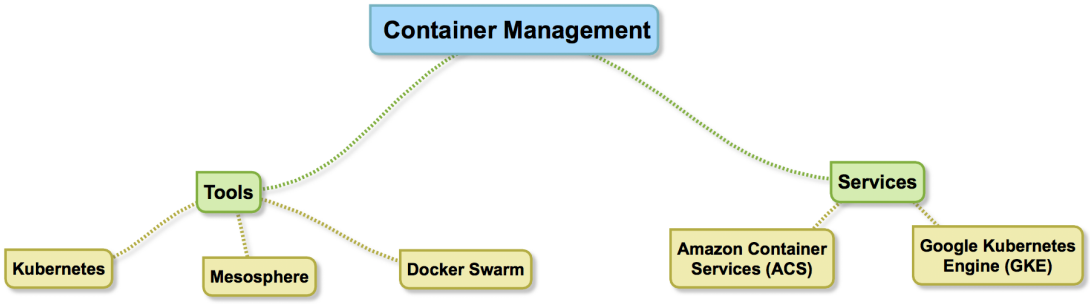

Figure 24: Classification of Container Management Solutions

The following table provides an overview of the most popular container management solutions:

	Docker Swarm	Kubernetes	Mesos Marathon	Google Kubernetes Engine	Amazon Container Services
Description	Extension of Docker to manage groups(swarms) of machines	A very popular Container Orchestration engine managed by CNCF.	Container Orchestration for Apache Mesos framework.	A closed source "premium" version of Kubernetes hosted on GCP	Native Container management on AWS cloud
Software Type	Open source	Open source	Open source	Closed Source	Closed Source
Support	Community	Community	Community	Google Cloud	AWS
Smallest Units of Deployment	Container	Pods	Groups	Pods	Tasks
Clustering	Container Based	Pod Based	Groups	Pods	Tasks
Storage Abstractions	Docker Volumes	Persistent Volumes (and claims)	Local Volumes (experimental Flocker integration)	Google storage integration	AWS storage integration
Networking concepts	Docker Networks	Network services	Limited support for port mapping	Provided by GCP	VPC on the AWS Cloud
Rolling Update support	Yes	Yes	Yes	Yes	Yes
ci/CD	Manually maintained	Addons available (Spinnaker)	Manually maintained	Addons available (Spinnaker)	AWS Code build / 3rd party services
Auto-scaling	Not available	Using Deployments/Replica sets	Experimental support	Using Deployments/Replica sets	Indirectly via cloudwatch alarms and lambda functions
Load Balancing	Swarm DNS (Not very scalable)	Ingress used for throttling, support for Native Load balancers	Mesos DNS out of the box, support for Native Load balancers	GCP Load Balancer	Amazon Elastic Load Balancer
Auto-Recovery	based on restart policy	based on restart policy	based on restart policy	based on restart policy	based on restart policy
Hybrid Cloud Support	Yes	Yes	Yes	No (GCP Only)	No (AWS Only)
Health Checking	None out of the box	Liveness, Readiness probes	Rest APIs from tasks	Liveness, Readiness probes	Cloudwatch based
Logging	Manually maintained	Prometheus support	Only via external tools	Google Stackdriver	Amazon Cloudwatch
Ops/Maintenance	High	High	High	Low	Low
Cost	VMs + Ops cost	VMs + Ops cost	VMs + Ops cost	One container cheaper than AWS*	Per resource used
Used in Production	Rarely	Yes	Not common	Yes	Yes

Figure 25: Comparison of Container Management Platforms