Computer Science and Engineering Department

Motilal Nehru National Institute of Technology Allahabad
Teliarganj, Allahabad, Uttar Pradesh 211004

A Light-weight Multi-tenant Distributed IoT platform using idle devices on a mesh network

Submitted in partial fulfillment of
the requirements for the award of the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by

Roll No	Names of Students
20144028	Harshit Pandey
20144061	Deepak Kumar Singh
20144099	Agrawal Shreyas Nanakchand
20144051	Jatin Rungta
20144118	Ashutosh Kumar Sonu

Under the guidance of
Dr. Shashwati Banerjea

Computer Science and Engineering Department
Motilal Nehru National Institute of Technology Allahabad
Teliarganj, Allahabad, Uttar Pradesh 211004

Computer Science and Engineering Department

Motilal Nehru National Institute of Technology Allahabad

UNDERTAKING

We declare that the work presented in this report titled '**A Light-weight Multi-tenant Distributed IoT platform using idle devices on a mesh network**', submitted to the Computer Science and Engineering Department, Motilal Nehru National Institute of Technology, Allahabad, for the award of the **Bachelor of Technology** degree in **Computer Science & Engineering**, is our original work. We have not plagiarized or submitted the same work for the award of any other degree. In case this undertaking is found incorrect, We accept that my degree may be unconditionally withdrawn.

Roll No	Names of Students	Signature
20144028	Harshit Pandey	
20144061	Deepak Kumar Singh	
20144099	Agrawal Shreyas Nanakchand	
20144051	Jatin Rungta	
20144118	Ashutosh Kumar Sonu	

Dr. Shashwati Banerjea
(Project Guide)

May, 2018

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad

CERTIFICATE

Certified that the work contained in the report titled 'A Light-weight Multi-tenant Distributed IoT platform using idle devices on a mesh network', has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Roll No	Names of Students
20144028	Harshit Pandey
20144061	Deepak Kumar Singh
20144099	Agrawal Shreyas Nanakchand
20144051	Jatin Rungta
20144118	Ashutosh Kumar Sonu

Dr. Shashwati Banerjea
(Project Guide)

May, 2018

PREFACE

We hereby present our project on the topic A Light-weight Multi-tenant Distributed IoT platform using idle devices on a mesh network. Our project deals with the problem of sending sensor data to an off-site cloud, causing higher round trip time in each processing cycle. We propose a fully automatic approach that utilizes idle nodes in the local network for the required computations and hence reducing the time taken to a great extent.

We also declare that we have taken help from the internet whose references have been stated in references. We have also taken immense help and guidance from our mentor Dr. Shashwati Banerjea and Dr.Mayank Pandey. We hope this thesis proves helpful to its readers.

May, 2018

ACKNOWLEDGEMENTS

First and foremost, we would like to convey our gratitude and gratefulness to our mentor Dr. Shashwati Banerjea and Dr. Mayank Pandey who guided us immensely in each step and helped us in every way possible to deliver the work we present. We would also like to convey our sincere gratitude to all those individuals on whose precious advice and guidance we were able to execute this feat.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Definition

Internet of Things (IoT) is a concept that visualizes all objects around us as part of internet. It's coverage is very wide and includes variety of objects like smart phones, digital cameras, sensors, etc. Generally all this collected data from the IOT sensor is sent to the cloud for the data computation (Figure 1.1). Cloud services require developers to host services, applications and data on off-site data centers.

Which means the computing and storage resources are relatively far away from the end user's devices. But due to application-specific reasons, a growing number of high quality services desire computational tasks to be located nearby. Now these days, number of research efforts have espoused the need and benefits of creating edge computing devices that distribute computational functions closer to the client devices.

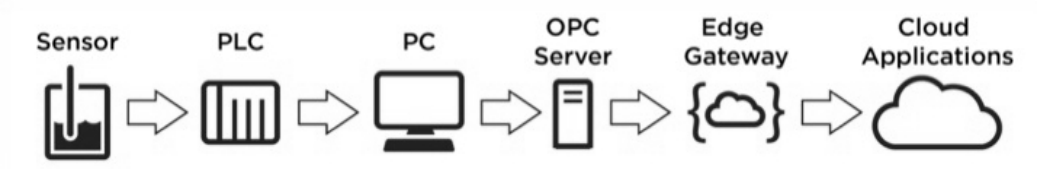

Figure 1.1: The Problem of multilayer latency

Hence, instead of sending data from devices or sensors to cloud for the computation , we are pushing the 'intelligence' to the edge of the network by doing the computation on idle devices in the same network.

Figure 1.2: IOT it's Cloud based Architecture

1.2 Background, recent development and use cases

The IoT owes its explosive growth to the connection of physical things and operation technologies (OT) to analytics and machine learning applications, which can help glean insights from device-generated data and enable devices to make 'smart' decisions without human intervention. Currently, such resources are mostly being provided by cloud service providers (Figure 1.2), where the computation and storage capacity exists.

However, despite its power, the cloud model is not applicable to environments where operations are time-critical or internet connectivity is poor. This is especially true in scenarios such as tele-medicine and patient care, where milliseconds can have fatal consequences. The same can be said about vehicle to vehicle communications, where the prevention of collisions and accidents cant afford the latency caused by the round trip to the cloud server. The cloud paradigm is like having your brain command your limbs from miles away it wont help you where you need quick reflexes.

Moreover, having every device connected to the cloud and sending raw data over the internet can have privacy, security and legal implications, especially when dealing with sensitive data that is subject to separate regulations in different countries.

The fog layer is the perfect junction where there are enough compute, storage and networking resources to mimic cloud capabilities at the edge and support the local ingestion of data and the quick turnaround of results.Fog computing improves efficiency and reduces the amount of data that needs to be sent to the cloud for processing.

IoT company Plat One is another firm using fog computing to improve data processing for the more than 1 million sensors it manages. The company uses the 'ParStream' platform to publish real-time sensor measurements for hundreds of thousands of devices.

1.3 Overview

1.3.1 Edge Computing

Edge computing [1] is a distributed information technology (IT) architecture in which client data is processed at the periphery of the network, as close to the originating source as possible. The move toward edge computing is driven by mobile computing, the decreasing cost of computer components and the sheer number of networked devices in the internet of things (IoT). Depending on the implementation, time-sensitive data in an edge computing architecture may be processed at the point of origin by an intelligent device or sent to an intermediary server located in close geographical proximity to the client. Data that is less time sensitive is sent to the cloud for historical analysis, big data analytics and long-term storage.

Transmitting massive amounts of raw data over a network puts tremendous load on network resources. In some cases, it is much more efficient to process data near its source and send only the data that has value over the network to a remote data center. Instead of continually broadcasting data about the oil level in a car's engine, for example, an automotive sensor might simply send summary data to a remote server on a periodic basis. Or a smart thermostat might only transmit data if the temperature rises or falls outside acceptable limits. Or an intelligent Wi-Fi security camera aimed at an elevator door might use edge analytics and only transmit data when a certain percentage of pixels significantly change between two consecutive images, indicating motion. A promising approach to addressing the challenges for data analytics in IoT is edge computing that pushes various computing and data analysis capabilities to network edges. Edge computing can enhance the reliability of the cloud, eliminating a bottleneck.

1.3.2 Distributed Storage Architecture

Storage plays a fundamental role in computing, a key element, ever present from registers and RAM to hard-drives and optical drives. Functionally, storage may service a range of requirements, from caching (expensive, volatile and fast) to archival (inexpensive, persistent and slow). Combining networking and storage has created a platform with numerous possibilities allowing Distributed Storage Systems[2] (or DSS) to adopt roles vast and varied which fall well beyond data storage. DSS are capable of spanning a global network of users providing a rich set of services; from publishing, file sharing and high performance to global federation and utility storage.

In a Peer-to-Peer architecture every node has the potential to behave as a server and a client, and join and leave as they wish. Routing continually adapts to what is an ever changing environment. Strengths of the Peer-to-Peer approach include resilience to outages, high scalability and an ability to service an unrestricted public user-base. These strengths vary depending on the category of Peer-to-Peer a system adopts.

There are three main categories of Peer-to-Peer architectures, Globally Centralized, Locally Centralized and Pure Peer-to-Peer (Figure 1.3). Each of these categories have a varying degree of centralization, from being globally centralized to locally centralized to having little or no centralization with pure Peer-to-Peer. One of the early Peer-to-Peer publishing packages, Napster [Oram 2001] is an example of a system employing a globally centralized architecture. Here, peers are required to contact a central server containing details of other peers and respective files. Unfortunately, this reliance on a globally central index server limits scalability and proves to be a Single Point of Failure (SPF).

The choice of architecture has a major influence on system functionality, determining operational boundaries and its effectiveness to operate in a particular environment. As well as functional aspects, the architecture also has a bearing on the mechanisms a system may employ to achieve consistency, routing and security. A centralized architecture is suited to controlled environments and while it may lack the scalability of its Peer-to-Peer counterpart, it has the ability to provide a consistent Quality of Service (QoS). By contrast a Peer-to-Peer architecture is naturally suited to a dynamic environment, key advantages include unparallelled scalability and the ability to adapt to a dynamic operating environment. Our discussion of architectures in this section has been presented in a chronological order. We can see that the evolution of architectures adopted by DSSs have gradually moved away

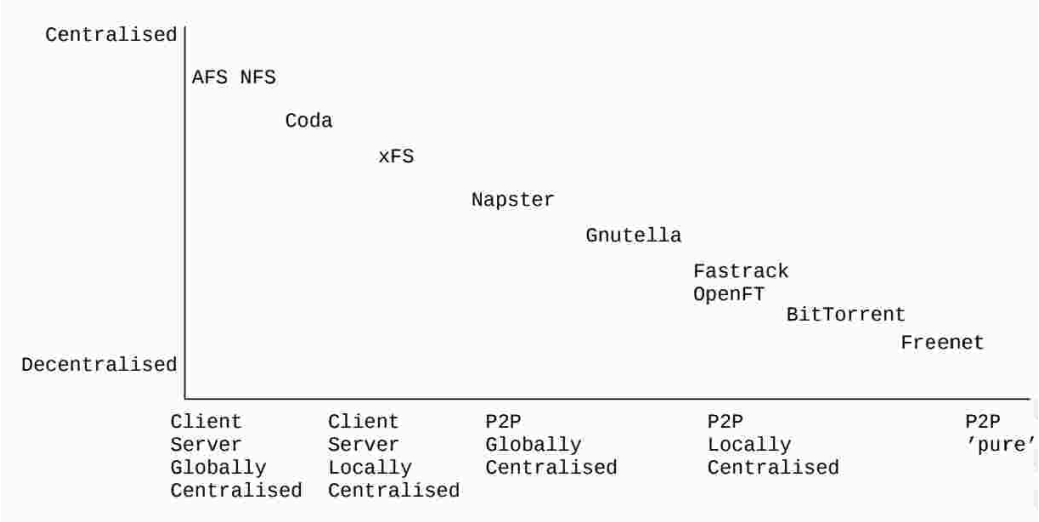

Figure 1.3: Storage Architecture Evolution

from centralized to more decentralized approaches (Figure 2.2), adapting to challenges associated with operating across a dynamic global network.

1.3.3 Docker

Docker [5] is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, the developer can assure that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code. Docker is a like a virtual machine. But unlike a virtual machine, rather than creating a whole virtual operating system, Docker allows applications to use the same Linux kernel as the system that they're running on and only requires applications be shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

Docker uses AuFS, a layered file system, so you can have a read only part and a write part which are merged together. One could have the common parts of the operating system as read only (and shared amongst all of your containers) and then give each container its own mount for writing.

So, let's say you have a 1GB container image; if you wanted to use a full VM, you would need to have 1GB times x number of VMs you want.

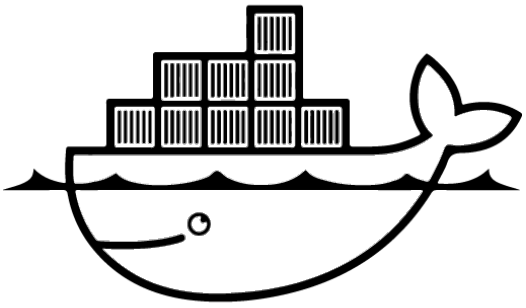

Figure 1.4: Docker Containers

With Docker and AuFS you can share the bulk of the 1GB between all the containers and if you have 1000 containers you still might only have a little over 1GB of space for the containers OS (assuming they are all running the same OS image).

Because Docker containers are so lightweight, a single server or virtual machine can run several containers simultaneously. By using containers, resources can be isolated, services restricted, and processes provisioned to have an almost completely private view of the operating system with their own process ID space, file system structure, and network interfaces. Multiple containers share the same kernel, but each container can be constrained to only use a defined amount of resources such as CPU, memory and I/O.

1.4 Probable Solutions

As mentioned in previous sections, we can clearly see that the major problems in the existing system are due to sending data to off-site processing units and not having a universal description format for the data and metadata.

We have attempted to solve these issues by :

- Using idle devices on a local network for processing the data.
- Creating a distributed architecture to overcome single point of failure and memory constraint of a single device.
- Creating a network monitoring interface to keep track of idle devices which can be used for processing incoming data.
- Creating an interface for data format conversion on the fly, which changes the format of incoming data to a preset universal format and thus helping achieve higher vendor interoperability.
- Deploying Docker containers on devices to do the processing to avoid operating system based dependencies.

Chapter 2

Related Work

2.1 Paradrop - Computation at the Extreme Edge

Internet of things data computation at extreme edge of network, the idea of our project is similar to one discussed in the research paper [6] "Peng Liu, Dale willis, Suman Banerjee, "Paradrop: Enabling Lightweight Multi-tenancy at the Network's Extreme Edge", in the University of Wisconsin Madison". The research paper has discussed about the Paradrop, a specific edge computing platform at the extreme edge of the network allowing the computation over the data generated by IoT sensors at the network's extreme edge. Paradrop framework has multiple key components:

- a virtualization substrate in the WiFi APs that the host third party computation in isolated containers (which we call, chutes)
- a cloud backend through which all of the paradrop APs and the third party containers are dynamically installed, instantiated, and revoked, and
- a developer API through which such developers can manage the resources of the platform and monitor the running status of APs and chutes.

Compared to other heavyweight counterparts, paradrop uses more efficient virtualization technology based on Linux containers rather than Virtual Machines(VMs), that means we can provide more resources for services with the given hardware. The virtualization substrate evolved from the original choice LXC to one based on Docker because of the latter is easy to use tools and wider adoption that would reduce the bar for developers.

2.2 Mobile Edge Computing (MEC)

Another similar idea is discussed in the research paper "Gabriel Orsini, Dirk Bade, Winfried Lamersdorf,"Computing at the Mobile Edge [7] : Designing Elastic Android Applications for Computation Offloading", in University of Hamburg, Germany.". It has discussed the idea of computing at the mobile edge. Current centralised cloud data centers provide scalable computation and storage resource in virtual infrastructure. But the current mobile devices and their resources-hungry application demand for these resource on the spot. Thus, mobile cloud computing was introduced to overcome thse limitation by transparently making accessible the apparently innite cloud resources to the mobile devices and by allowing mobile applications to (elastically) expand into the cloud. However, MCC often relies on a stable and fast connection to the mobile devices surrogate in the cloud, which iis a rare case in mobile scenarios. Moreover, the increased latency and the limited bandwidth prevent the use of real-time applications like, e.g. cloud gaming. Instead, mobile edge computing (MEC) or fog computing tries to provide the necessary resources at the logical edge of the network by including infrastructure components to create ad-hoc mobile clouds. However, this approach requires the replication and management of the applications business logic in an untrusted, unreliable and constantly changing environment. Consequently, this paper presents a novel approach to allow mobile app developers to easily benet from the features of MEC.

2.3 An optimization model of load balancing in Peer to Peer (P2P) Network

In this work, an optimization model of load balancing in P2P network is proposed to make full use of each node resource, and to consider the coordination of network resource allocation between the other network area and this area, making the whole network resource can be used as fully as possible, based on the load information of the different nodes, we layer the nodes and use an undirected graph to link each node. We put the ratio that come from the adding two nodes load capacity divided by the volume of the optimal load between the adjacent nodes as the strength of the road weights. When node load need several nodes to finish at the same time, find the optimal path of transfer load doing load transfer. And in need not considering several nodes load transfer, we can accord to the information of node list record to load or to join the node. It can optimize the load balancing in Peer to Peer Network and achieve high speed, it also can reduce the cost of load delay.

Chapter 3

Proposed Architecture

Cloud computing platforms, such as Amazon EC2, Microsoft Azure and Google App Engine have become a popular approach to providing ubiquitous access to services across different user devices. These platforms provide high-quality services to their end-users since they are reliable, always on and robust. But the flaw in this approach is that the computing and storage resources are relatively far away from end-users devices and thus require an overhead task of sending raw data and receiving back processed data over the internet, exposing end users' private data to the world. Network bandwidth acts as a bottleneck for such an arrangement as the round trip time from the cloud depends largely on network traffic.

Due to application-specific reasons, a growing number of high-quality services desire computational tasks to be located nearby. They include needs for lower latency, greater responsiveness, a better end user experience, and more efficient use of network bandwidth. As a consequence, over the last decade, a number of research efforts have espoused the need and benefits of creating Edge Computing services that distribute computational functions closer to the client devices.

This project is basically built on the above idea of doing the processing of data as close to the data source as possible using the idle devices of the network as our processing machines. These machines do the computation on data and then return the result to the respective client. For processing environment on the nodes, we had two choices - Virtual Machines or Docker Containers. Seeing the benefits of Docker containers over Virtual Machines in terms of resource requirement and ease of maintenance we chose them. A log of IP: CPU usage and capabilities is maintained for all the volunteer machines on the network in a distributed hash table among the nodes which helps us in choosing the machine with minimum CPU usage for our computation.

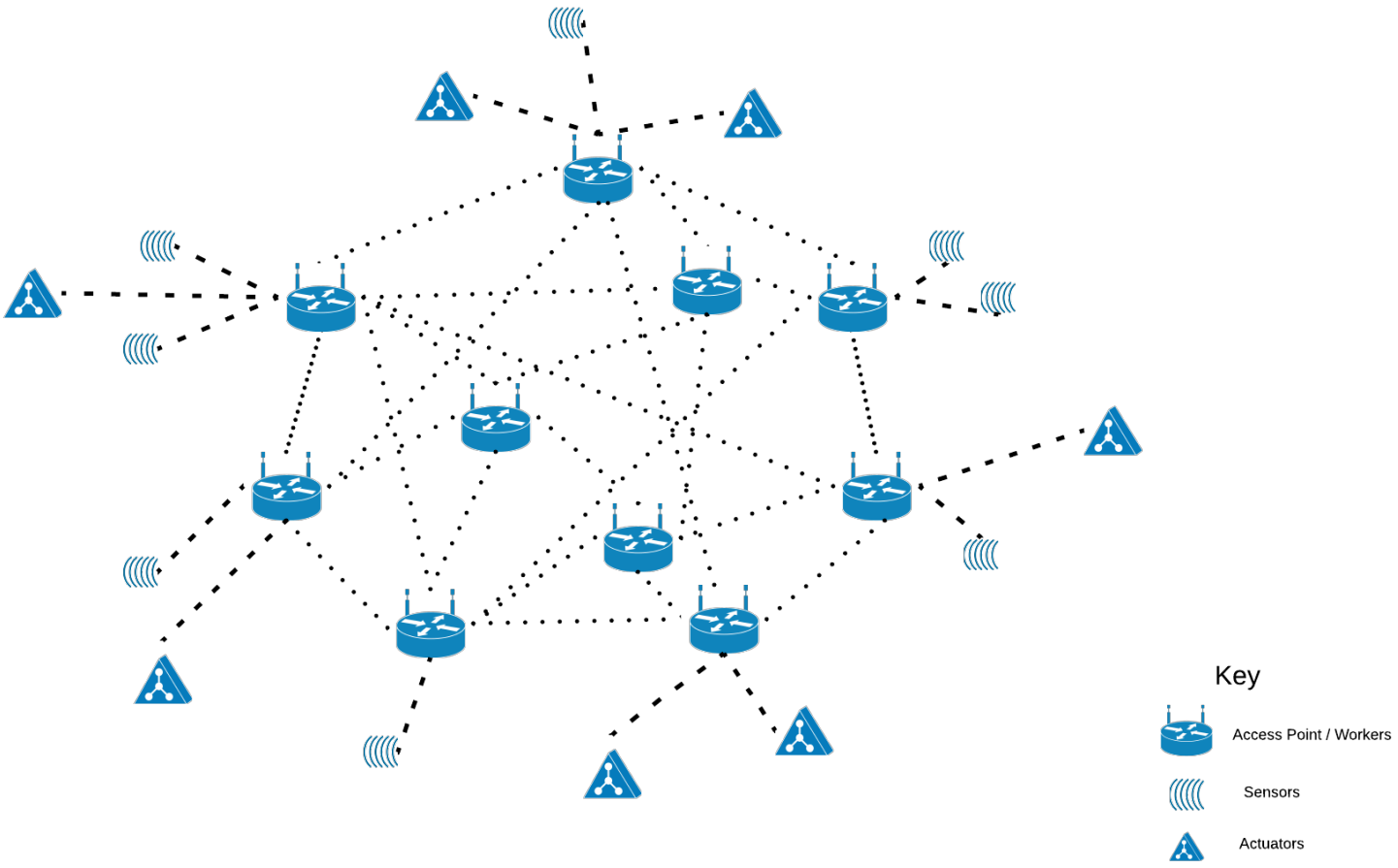

Figure 3.1: Proposed Architecture

3.1 Implementation Details

All the processing nodes run two modules,

- sensorserver.py
- worker.py
- framework translation module

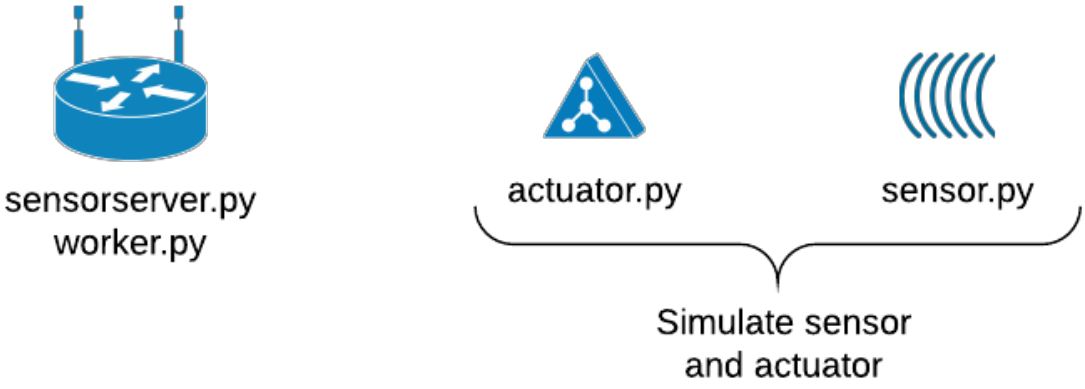

Figure 3.2: Modules

3.1.1 Senserserver Module

The first node running the sensorserver module becomes the bootstrap node for the distributed hash table. It launches a DHT instance and creates objects for storing worker capabilities and their cpu usage.

Any other node running the sensorserver module requests the bootstrap node to become a part of the existing DHT instance. Thus all nodes running sensorserver create a virtual storage ring. Any node amongst them can access the DHT data.

Sensorserver module also actively listens to data coming from sensors and route it to worker with the least cpu usage at that instant. Each member of the DHT keeps on updating it's avg cpu usage data in the shared table through which the most idle system is selected.

3.1.2 Worker Module

The worker module actively listens on a port to receive any incoming task from any sensorserver. On receiving a sensor data, the worker invokes a docker container instance corresponding to that particular sensor and does the required computations there in.

The fetched result is sent to the desired actuator for taking required decision.

3.1.3 Framework Translation module

As a part of the sensorserver module, this module changes the received data,metadata package from the sensors' description format to a predefined universal resource description format.

For example, if the sensor packs data in XML or any json format, the module extracts information from this packet and restructures it into a preset JSON format.

This helps in achieving multiple vendor compatibility without designing separate module for each vendor. Thus helps in high order scalability.

3.2 Distributed System Architecture

3.2.1 Distributed v/s Centralized approach

Maintaining a central server for storage of data is although a simpler approach but it leads to problems such as single point of failure and data faults. Thus, to prevent such issues we implemented a distributed storage architecture using storage of all participating nodes of the network by implementing a Distributed Hash Table.

A distributed hash table (or DHT[4]) is a class of a decentralized distributed system that provides a look-up service similar to a hash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of

disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. DHT's characteristically emphasize the following properties:

- Autonomy and decentralization: the nodes collectively form the system without any central coordination.
- Fault tolerance: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing.
- Scalability: the system should function efficiently even with thousands or millions of nodes.

A distributed hash table provides incremental scalability of throughput and data capacity as more nodes are added to the cluster. To achieve this, we horizontally partition tables (Table Sharding) to spread operations and data across bricks. Each brick thus stores some number of partitions of each table in the system, and when new nodes are added to the cluster, this partitioning is altered so that data is spread onto the new node. This horizontal partitioning evenly spreads both load and data across the cluster.

3.2.2 Data Recovery in case of node failure

Given that the data in the hash table is spread across multiple nodes, if any of those nodes fail, then a portion of the hash table will become unavailable. For this reason, each partition in the hash table is replicated on more than one cluster node. The set of replicas for a partition form a replica group; all replicas in the group are kept strictly coherent with each other. Any replica can be used to service a **get()**, but all replicas must be updated during a **put()** or **remove()**. If a node fails, the data from its partitions is available on the surviving members of the partitions' replica groups. Replica group membership is thus dynamic; when a node fails, all of its replicas are removed from their replica groups. When a node joins the cluster, it may be added to the replica groups of some partitions (such as in the case of recovery, described later).

To maintain consistency when state changing operations (put() and remove()) are issued against a partition, all replicas of that partition must be synchronously updated. We use an optimistic two-phase commit protocol to achieve consistency, with the DDS library serving as the commit coordinator and the replicas serving as the participants. If the DDS library crashes after

prepare messages are sent, but before any commit messages are sent, the replicas will time out and abort the operation.

If a brick fails, all replicas on it become unavailable. Rather than making these partitions unavailable, we remove the failed brick from all replica groups and allow operations to continue on the surviving replicas. When the failed brick recovers (or an alternative brick is selected to replace it), it must *catch up* to all of the operations it missed. In many RDBMS's and file systems, recovery is a complex process that involves replaying logs, but in our system we use properties of clusters and our DDS design for vast simplifications.

Firstly, we allow our hash table to say no; *i.e* bricks may return a failure for an operation, such as when two puts() to the same key are simultaneously issued, or when replica group memberships change during an operation. The freedom to say no greatly simplifies system logic, since we don't worry about correctly handling operations in these rare situations. Instead, we rely on the DDS library to retry the operation. Secondly, we don't allow any operation to finish unless all participating components agree on the metadata maps. If any component has an out-of-date map, operations fail until the maps are reconciled.

We make our partitions relatively small (approximately 100MB), which means that we can transfer an entire partition over a fast system-area network (typically 100 Mb/s to 1 Gb/s) within 1 to 10 seconds. Thus, during recovery, we can incrementally copy entire partitions to the recovering node, obviating the need for the undo and redo logs that are typically maintained by databases for recovery. When a node initiates recovery, it grabs a write lease on one replica group member from the partition that it is joining; this write lease means that all state-changing operations on that partition will start to fail. Next, the recovering node copies the entire replica over the network. Then, it sends updates to the RG map to all other replicas in the group, which means that DDS libraries will start to lazily receive this update. Finally, it releases the write lock, which means that the previously failed operations will succeed on retry. The recovery of the partition is now complete, and the recovering node can begin recovery of other partitions as necessary.

There is an interesting choice of the rate at which partitions are transferred over the network during recovery. If this rate is fast, then the involved bricks will suffer a loss in read throughput during the recovery. If this rate is slow, then the bricks won't lose throughput, but the partition's mean time to recovery will increase. We chose to recover as quickly as possible, since in a large cluster only a small fraction of the total throughput of the cluster

will be affected by the recovery.

3.3 Kademlia DHT

Kademlia[3] is a distributed hash table for decentralized peer-to-peer computer networks designed by Petar Maymounkov and David Mazires in 2002. It specifies the structure of the network and the exchange of information through node lookups. Kademlia nodes communicate among themselves using UDP. A virtual or overlay network is formed by the participant nodes. Each node is identified by a number or node ID. The node ID serves not only as identification, but the Kademlia algorithm uses the node ID to locate values (usually file hashes or keywords). In fact, the node ID provides a direct map to file hashes and that node stores information on where to obtain the file or resource.

When searching for some value, the algorithm needs to know the associated key and explores the network in several steps. Each step will find nodes that are closer to the key until the contacted node returns the value or no more closer nodes are found. This is very efficient: Like many other DHTs, Kademlia contacts only O(log(n)) nodes during the search out of a total of n nodes in the system.

Further advantages are found particularly in the decentralized structure, which increases the resistance against a denial-of-service attack. Even if a whole set of nodes is flooded, this will have limited effect on network availability, since the network will recover itself by knitting the network around these "holes".

Kademlia routing tables consist of a list for each bit of the node ID. (e.g. if a node ID consists of 128 bits, a node will keep 128 such lists.) A list has many entries. Every entry in a list holds the necessary data to locate another node. The data in each list entry is typically the IP address, port, and node ID of another node. Every list corresponds to a specific distance from the node. Nodes that can go in the nth list must have a differing nth bit from the node's ID; the first n-1 bits of the candidate ID must match those of the node's ID. This means that it is very easy to populate the first list as 1/2 of the nodes in the network are far away candidates. The next list can use only 1/4 of the nodes in the network (one bit closer than the first), etc.

With an ID of 128 bits, every node in the network will classify other nodes in one of 128 different distances, one specific distance per bit.As nodes are encountered on the network, they are added to the lists. This includes store and retrieval operations and even helping other nodes to find a key. Every

node encountered will be considered for inclusion in the lists. Therefore, the knowledge that a node has of the network is very dynamic. This keeps the network constantly updated and adds resilience to failures or attacks.

Kademlia nodes are represented in the form of a binary tree where nodes are the leaves of the binary tree, [Figure 3.3]

Kademlia uses four messages.

- PING used to verify that a node is still alive.
- STORE Stores a (key, value) pair in one node.
- FIND_NODE The recipient of the request will return the k nodes in his own buckets that are the closest ones to the requested key.
- FIND_VALUE Same as FIND_NODE, but if the recipient of the request has the requested key in its store, it will return the corresponding value.

Each RPC message includes a random value from the initiator. This ensures that when the response is received it corresponds to the request previously sent.

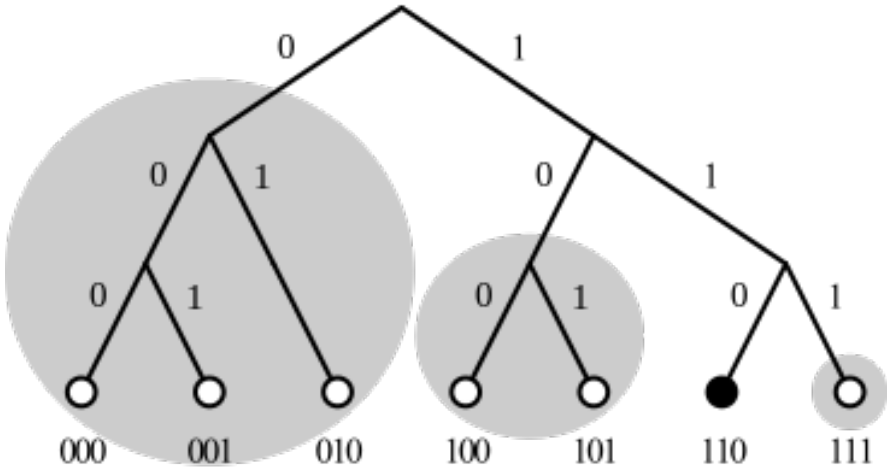

Figure 3.3: Binary tree structure of Kademlia Nodes

3.3.1 Why Kademlia?

Kademlia[3] provides many desirable features that are not simultaneously offered by any other DHT [4]. Kademlia has several properties that make it a preferred choice of DHT. These include:

- Kademlia minimizes the number of inter-node introduction messages.
- Configuration information such as nodes on the network and neighboring nodes spread automatically as a side effect of key lookups.
- Nodes in Kademlia are knowledgeable of other nodes. This allows routing queries through low latency paths.
- Kademlia uses parallel and asynchronous queries which avoid timeout delays from failed nodes.
- Kademlia is resistant to some DOS attacks.

3.4 Network Monitoring

As soon as any end device sends in data to be processed, the parent node searches for all the nodes in the network which have the resources to successfully process that data. For this we maintain a Distributed Hash Table entry in form :

$$[Resource, \langle listofnodes \rangle]$$

Thus a query for resource search returns a list of nodes holding it. To keep a track on the cpu usage of all systems in a network backbone to figure out idle systems, we developed a Distributed Hash Table over the existing P2P system which saves CPU usage of all systems in the backbone and maintain an (IPaddress, CpuUsage) Key-Value entry, which enables us to select the most idle system on the network and utilize it's computational power to process data generated by end devices.

3.5 Processing of data at idle systems

After selecting a node with the required resource and with minimum cpu usage, the main task for the parent node is to process the sensor's data on it by running required scripts. After discussions over various available platforms, we chose to deploy Docker containers on the client to do the processing.

3.6 Docker Container vs Virtual Machine instance

There are pros and cons for each type of virtualized system. If you want full isolation with guaranteed resources, a full VM is the way to go. If you just want to isolate processes from each other and want to run a ton of them on a reasonably sized host, then Docker/LXC/runC seems to be the way to go.

Docker gives you the ability to snapshot the OS into a shared image, and makes it easy to deploy on other Docker hosts. This gives us the flexibility to enable multi tenancy to our proposed systems as vendors only need to provide us an encrypted docker image of their proprietary processing software.A local Docker Repository is hosted through which slave nodes can obtain the requisite image to process the data they receive.

Docker ensures your applications and resources are isolated and segregated. A few months back, Gartner published a report stating Docker containers are as good as VM hypervisors when it comes to isolating resources, but there is still work to be done in terms of management and administration.

Consider a scenario where you are running multiple applications on your VM. These applications can be team collaboration software (e.g., Confluence), issue tracking software (e.g., JIRA), centralized identity management systems (e.g., Crowd) and so on. Seeing as all of these applications run on different ports, you would have to leverage them on Apache and Nginx as a reverse proxy. So far, everything is in good shape, but as your environment moves forward, you will also need to configure a content management system (e.g., Alfresco) into your existing environment. Bear in mind that it requires a different version of Apache Tomcat, which will cause a problem. In order to fix this, you can either move your existing applications to another version of Tomcat or run your content management system (Alfresco) on your currently deployed version.

Fortunately, with Docker, you dont have to do this. Docker makes sure each container has its own resources that are isolated from other containers. You can have various containers for separate applications running completely different stacks. Aside from this, effectively removing applications from your server is quite difficult and may cause conflicts with dependencies. However, Docker helps you ensure clean app removal since each application runs on its own container. If you no longer need an application, you can simply delete its container. It wont leave any temporary or configuration files on your host OS.

3.6.1 Why Docker?

Docker [5] is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating systemlevel virtualization on Linux.

A full virtualized system gets its own set of resources allocated to it, and does minimal sharing. You get more isolation, but it is much heavier (requires more resources). With Docker you get less isolation, but the containers are lightweight (require fewer resources). So you could easily run thousands of containers on a host, and it won't even blink.A full virtualized system usually takes minutes to start, whereas Docker/LXC/runC containers take seconds, and often even less than a second, which makes Docker containers perfect fit for our purpose as IoT demands lighting fast processing.

3.7 Using Universal Resource Description Framework

Interoperability challenges can arise when attempting to integrate devices made by different manufacturers. If the device and cloud service are from the same vendor and if proprietary data protocols are used between device and cloud service, then user may be tied to a specific cloud service, limiting or preventing the use of alternative service providers. This is commonly referred to as vendor lock-in. IoT customers too are starting to consider Vendor lock in problem.

Each vendor develops their own method of packaging metadata and data in a network packet. Thus it becomes difficult to create a system that handles all vendors as it would require separate modules for separate vendors, which will increase the complexity of the system.

The Resource Description Framework (RDF) is an infrastructure that enables the encoding, exchange and reuse of structured metadata. RDF is an application of XML (or JSON) that imposes needed structural constraints to provide unambiguous methods of expressing semantics. The structural constraints RDF imposes to support the consistent encoding and exchange of standardized metadata provides for the interchangeability of separate packages of metadata defined by different resource description communities.

We created the solution to these vendor specific needs by creating a Universal Resource Description Format. The incoming packets are unpacked and the received ¡data,metadata¿ is packaged again into the designed description format on the fly. By this we eliminated the need of vendor specific modules and created a single module that handles the universal format specified.

Chapter 4

Results

4.1 Comparison with centralized server model

We compared our architecture with a generic central server task distribution model on the following parameters :

- Total Processing Delay (in seconds)
- Total Transmission delay (in milliseconds)

Data was collected for set of 5,10,15,20,25 and 30 nodes and taking the average value over 1000 data points for each of them.

Number of Nodes	5	10	15	20	25	30
Centralized system	1.03	2.1	2.56	2.97	3.23	3.76
Decentralized system	1.1	1.45	1.57	1.74	1.91	1.97

Comparison of total processing delay

Number of Nodes	5	10	15	20	25	30
Centralized system	10.8	14.9	17.3	18.8	20.2	23.61
Decentralized system	12.1	16.73	17.1	18.3	19.7	21.5

Comparison of total transmission latency

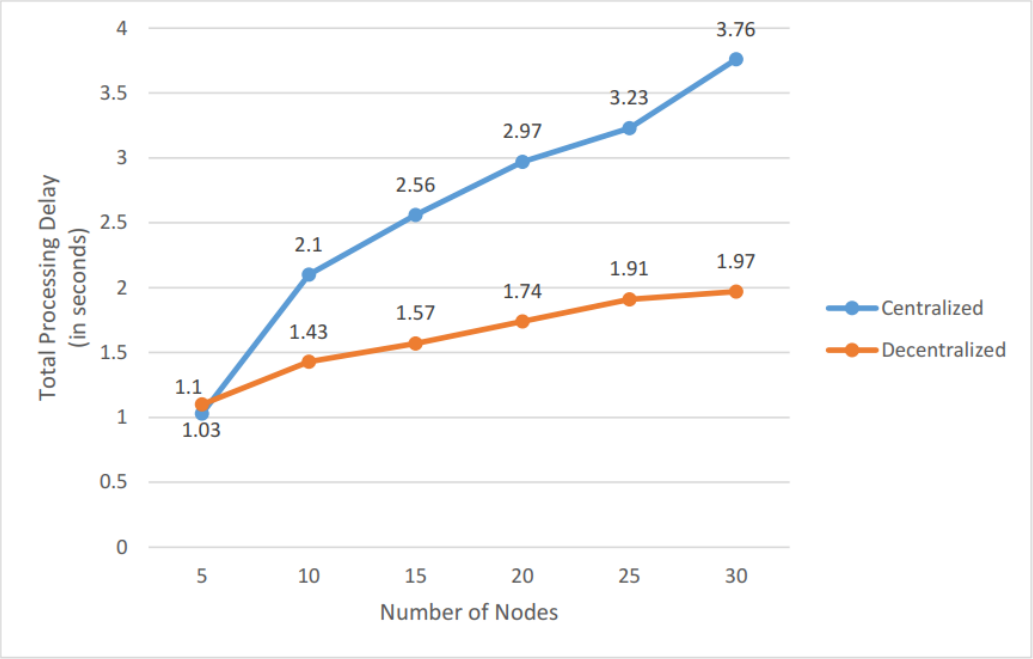

Figure 4.1: Comparison of total processing delay

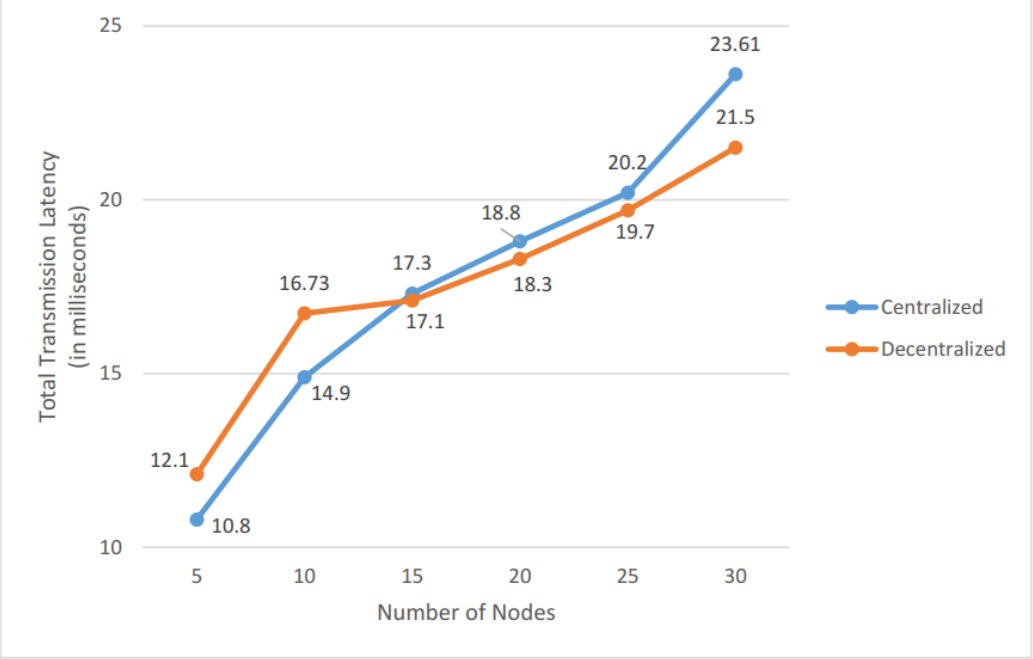

Figure 4.2: Comparison of total transmission latency

Chapter 5

Challenges faced and Plausible Future Extensions

5.1 Challenges faced

5.1.1 Selecting a node to work

From a pool of computers having the computation ability to work on the received data from sensors, we need to select the idlest CPU. We were selecting the node solely based on CPU usage at the particular instant. We did not consider the effects of blocked processes and the fact that CPU usage could change at any instant making the criteria a poor choice for a metric. We then considered load average which takes into account the blocked process and the number of cores along with CPU usage over a period of 1 minute making it a stable and a reliable metric over CPU usage.

5.1.2 Space constraints on the nodes

The docker image we created was based on Debian along with python and weighed over 700 Mb. We were deploying a whole OS just to compute our data. The test computers did have a lot of space available on their disks but the size of the image was unacceptable. We do not need to pack functionality that we would never have used. We chose to go for an alpine based image that trimmed down the image size to just 80Mb making it feasible to deploy on space constrained nodes.

5.1.3 Authenticity of Sensor

We assumed that we would not get any stray traffic over the network and thus did not check the authenticity of the sensor or the data received over the network. When the program received undesirable data, the server would not be able to process this request as this was not handled. We are still working on this issue.

5.1.4 Processing speed v/s Sending speed

To optimize the use of network resources, we are creating only a single connection between the server and a sensor. This connection has a single buffer. The time taken to process the data (choosing a worker, computing on data and sending back to the actuator) can become more than the time required for the sensor to send another data. Since a connection is using buffers, data over multiple sends would accumulate at the buffer causing the next processing to be done over combined and thus incorrect data resulting in an error. We thus decided to use different connections between the server and a sensor for each data set to fix the problem temporarily.This is not the efficient use of network resources. We are working to strike a balance between the processing and the sending of the data.

5.2 Plausible Future Extensions

5.2.1 Increasing the robustness of system

Current systems requires a handful of dependencies to run. Thus reducing the number of dependencies required or creating a wrapper for all dependencies is of prime importance at the current stage.

5.2.2 Common interface handling all protocol stacks

Create complete locality and obtain ease of working with different sensor vendors by creating a common interface for all protocol stacks.

5.2.3 Docker Swarm to deploy services

Implement the same system using Docker Swarm as it comes with several built-in features such as Load-Balancing and handling of node failure.

5.2.4 Cross platform compatibility

Include cross platform compatibility for iOS based systems.

5.2.5 Creation of Clusters for distributed processing

We plan to implement a system for creation of cluster using several idle nodes which can be used to process tasks whose requirements are more than those present at any single node.

5.2.6 Implementing a local Docker Repository

In the present version, if it happens to be that the required docker image resource is unavailable at any of the local nodes, then the system would use the primitive path of requesting the image from a global repository. In future versions we tend to implement a local repository on one of the nodes not participating in processing which will increase the speed of these fetch operations to a great extent.

5.2.7 Phantom Nodes

Currently we are working on the assumption that nodes do not power down any time.If it happens there would be a false entry against that nodes id in the DHT and it might happen that it is selected for processing based on that invalid data. This will lead to connection failure and hence data loss. In its present version, our DHT implementation doesn't allow key deletion, which we plan to implement in our next version.

References

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li and Lanyu Xu. *Edge Computing: Vision and Challenges.* [*Wayne State University*]

[2] Martin Placek and Rajkumar Buyya. *A Taxonomy of Distributed Storage Systems.* [*The University of Melbourne*] http://www.cloudbus.org/reports/DistributedStorageTaxonomy.pdf

[3] Petar Maymounkov and David Mazires. *Kademlia - A P2P information system based on the XOR metric.* [*New York University*] https://kademlia.scs.cs.nyu.edu

[4] Distributed Hash Tables http://www.linuxjournal.com/article/6797

[5] Docker https://www.docker.com/

[6] Peng Liu, Dale Willis and Suman Banerjee. *Enabling Lightweight Multi-tenancy at the Networks Extreme Edge* [*University of Wisconsin-Madisons*]

[7] Gabreil Orsini et al.. *Computing at the Mobile Edge*[*Designing Elastic Application for Computation Offloading*]. 8th IFIP Wireless and Mobile Networking Conference,2015