Week 3 Lecture 1

0 mins

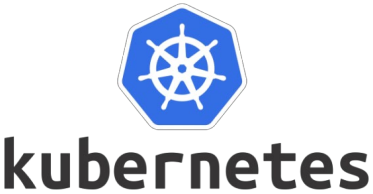
Kubernetes

Dr. Rajiv Misra, Professor
Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Contents

- Introduction to Kubernetes
- Why is Kubernetes required?
- Components of Kubernetes
- Deploying Applications with Kubernetes

Introduction to Kubernetes



- Kubernetes is the Greek word for helmsman or captain of a ship.
- Kubernetes also known as **k8s** was built by Google based on their experience running containers in production
- Kubernetes is an open-source **container orchestration** platform.
- **What is Container Orchestration?**
- it is the overall process of managing the lifecycle of containers that usually run in a production environment.
- **Why is Container Orchestration required?**
- Due to the rise of micro-service architecture, the use of containerization increased tremendously.
- Large applications spanning across multiple containers needs to be managed.

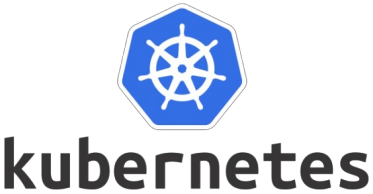
Transcript Notes

- The name Kubernetes originates from Greek, meaning helmsman or pilot.
- K8s as an abbreviation results from counting the eight letters between the "K" and the "s".
- Kubernetes is an open-source **container orchestration** platform by Google (open-sourced in 2014)

- **What is Container Orchestration?** – it is the overall process of managing the lifecycle of containers that usually run in a production environment.
- These containers, for example, may be deployed by an organization to host its applications (e.g., web-servers, e-mail servers, business-services etc.) which earlier used to be hosted on Virtual Machines or dedicated physical servers.
- **Why is Container Orchestration required?**
- As we have seen previously, Modern Application development heavily favors the micro-services architecture.
- and due to the rise of micro-service architecture, the use of containerization has increased tremendously
- the micro-services are now being provided in a containerized form since it becomes easy to develop, deploy and manage them independently.
- furthermore, containers provide greater benefits over hosting applications on a virtual machine or on the physical servers.
- Eventually this resulted in large-applications to grow and span multiple containers deployed across **multiple servers or nodes**, so operating them became more complex.
- To manage this complexity, some way of “orchestration” was required.

Introduction to Kubernetes

- Kubernetes takes this approach of clustering nodes and orchestrating containers on a cluster level.
- Container orchestration includes a range of functions, including but not limited to:
  - **Provisioning**— that installing or upgrading the local operating system on a node or setting up the container runtime there.
  - **Organizational primitives**, such as labels in Kubernetes, querying the containers, grouping the containers
  - **Scheduling** of containers to run on a host (one of the nodes in a cluster)
  - **Automated health checks** to determine if a container is alive and ready to serve traffic and to relaunch it if necessary
  - **Autoscaling** (that is, increasing or decreasing the number of containers based on utilization or higher-level metrics)
  - **Upgrade strategies**, proving rolling updates and rollback mechanism
  - **Service discovery** to determine which host a scheduled container ended upon, usually including DNS support



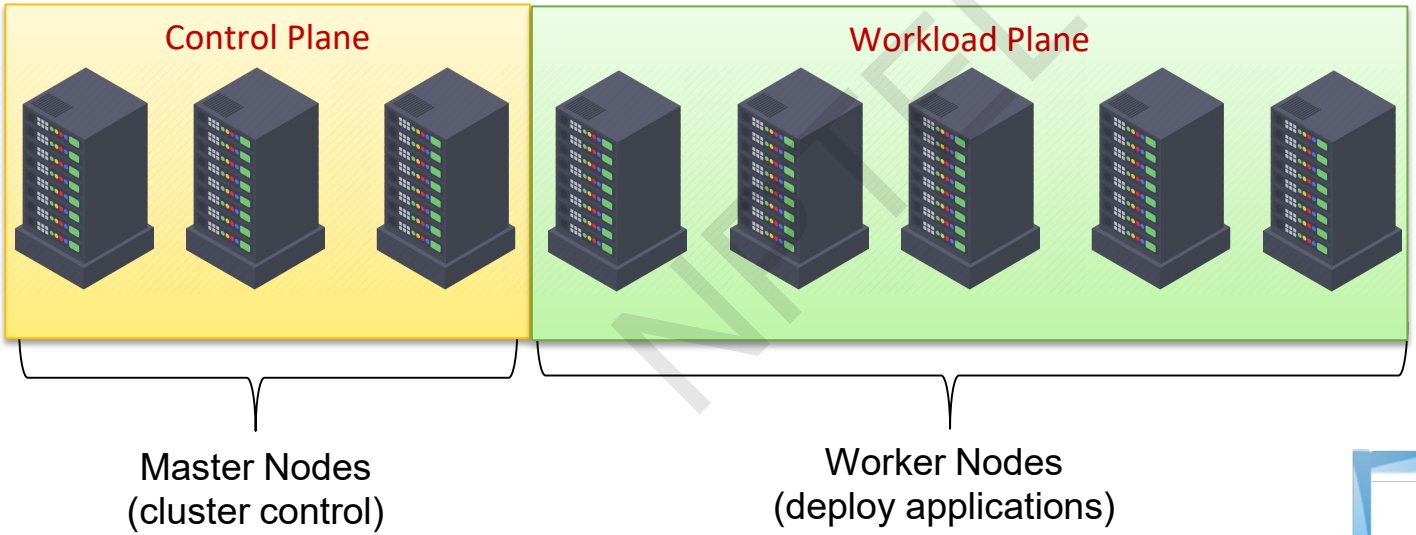
Transcript Notes

- Often, the production environments where the applications are hosted, can **have multiple servers or nodes** that can be organized in **clusters.**
- Kubernetes takes this approach of clustering nodes and orchestrating containers on a cluster level.

- **Container orchestration** includes a range of functions, for example,
  - **Provisioning**— that installing or upgrading the local operating system on a node or setting up the container runtime there.
  - **Organizational primitives**, such as labels in Kubernetes, querying the containers, grouping the containers
  - **Scheduling** of containers to run on a host (one of the nodes in a cluster)
  - **Automated health checks** to determine if a container is alive and ready to serve traffic and to relaunch it if necessary
    - **self-healing** of applications by automatically restarting or replicating containers.
  - **Autoscaling** (that is, increasing or decreasing the number of containers based on utilization or higher-level metrics)
    - **horizontal scaling** - replicating application containers to meet the increased demand
    - **load balancing** – distributing workloads across multiple replicas of the application containers
  - **Upgrade strategies**, proving rolling updates and rollback mechanism
  - **Service discovery** to determine which host a scheduled container ended upon, usually including DNS support

Kubernetes Cluster

Kubernetes cluster



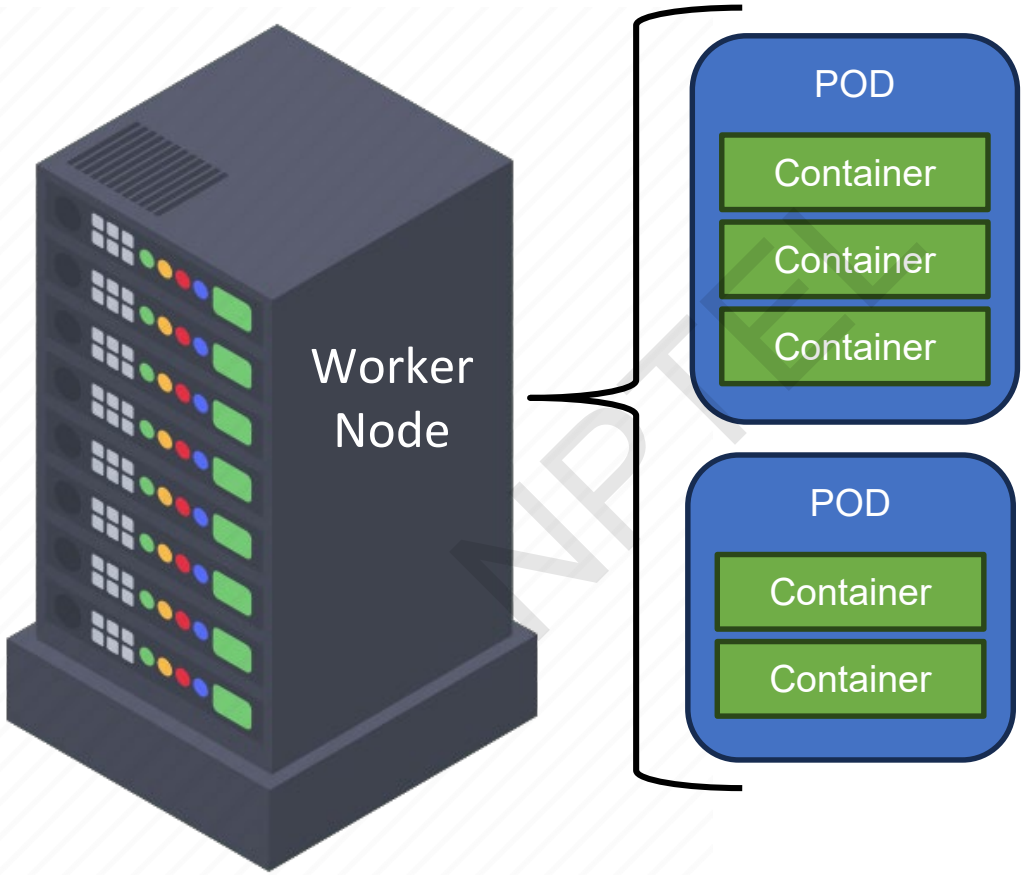
Transcript Notes

Understanding working of Kubernetes

The Kubernetes architecture follows a **master-worker model**, where the master, known as the control plane, manages the worker nodes. On the other hand, containers are deployed and executed in the worker nodes. These nodes can have virtual machines (on-premise or on the cloud) or physical servers.

- As shown in the figure, **A Kubernetes cluster** is a collection of nodes that are divided into 2 groups - the **master** and the **worker** nodes.
- The **master nodes** will run the Kubernetes **Control Plane**, which controls the cluster, while the **worker nodes** will run applications - and will therefore represent the **Workload Plane**. (The Workload Plane is sometimes referred to as the Data Plane,)
- The **Control Plane** is responsible for managing the state of the cluster. In a production environment, the control plane may span across multiple nodes and even multiple data centers. The **Workload Plane or Data Plane** contains the worker nodes that run the containerized application workloads. The containerized applications run in a **Pod**.

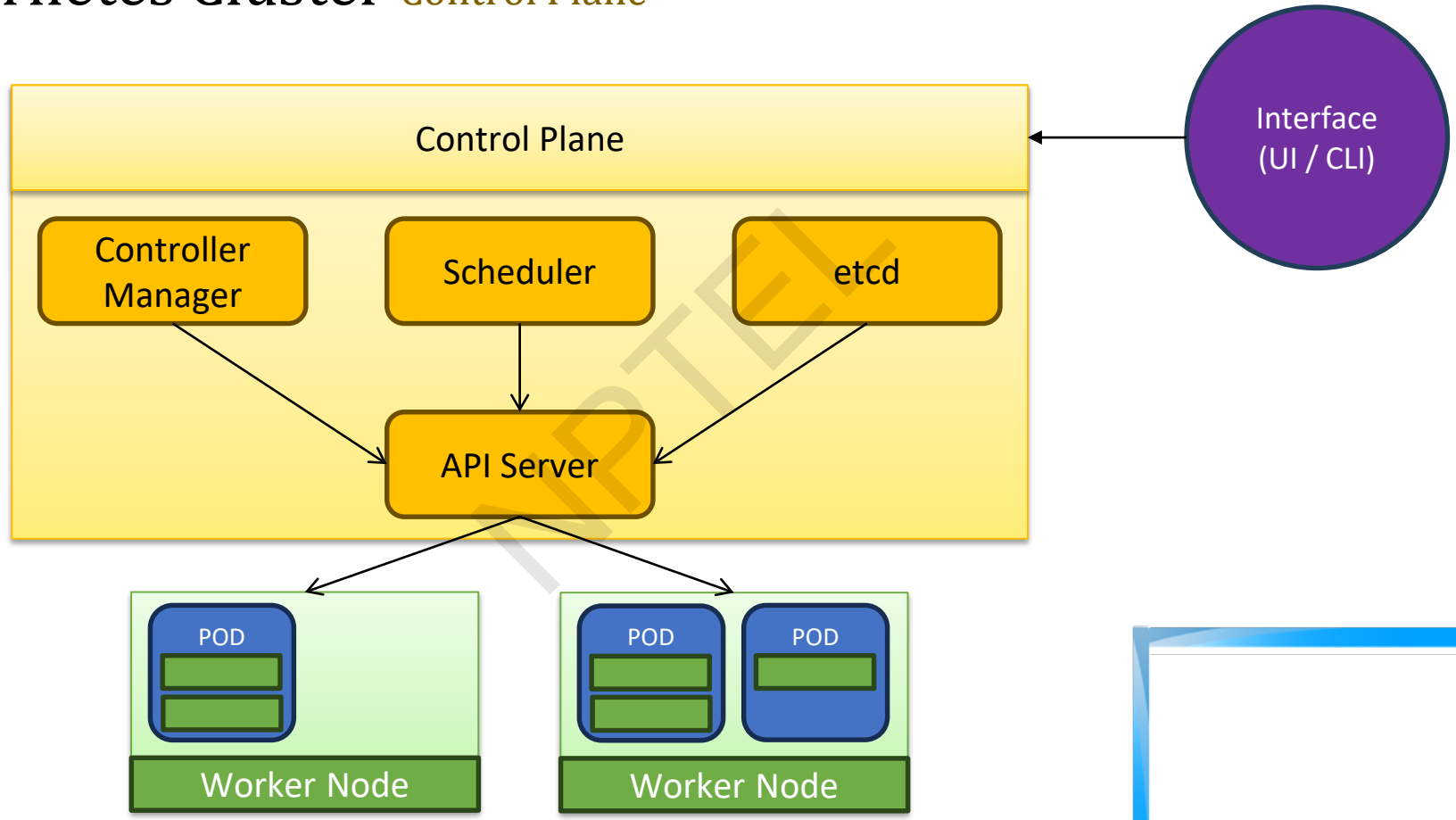
Kubernetes Cluster Pods



Transcript Notes

- Pods are the smallest deployable units in Kubernetes. They are the basic building blocks of Kubernetes applications.
- A pod hosts one or more containers and provides shared storage and networking for those containers.
- Therefore, Pods represents the set of running containers in a cluster.
- Pods are created and managed by the Kubernetes **control plane.**

Kubernetes Cluster Control Plane



Transcript Notes

- Overview of Control Plane

- Control Plane consists of a number of core components.
- They are the **API server**, **etcd**, **scheduler**, and the **controller manager**.

- **The API server** is the primary interface between the control plane and the rest of the cluster.
- It exposes a RESTful API that allows clients to interact with the control plane and submit requests to manage the cluster.
- for e.g., **kubectl (cube-CTL)** is a **command line interface** that can be used to manage Kubernetes clusters. It use HTTP-REST APIs to talk to the API server.

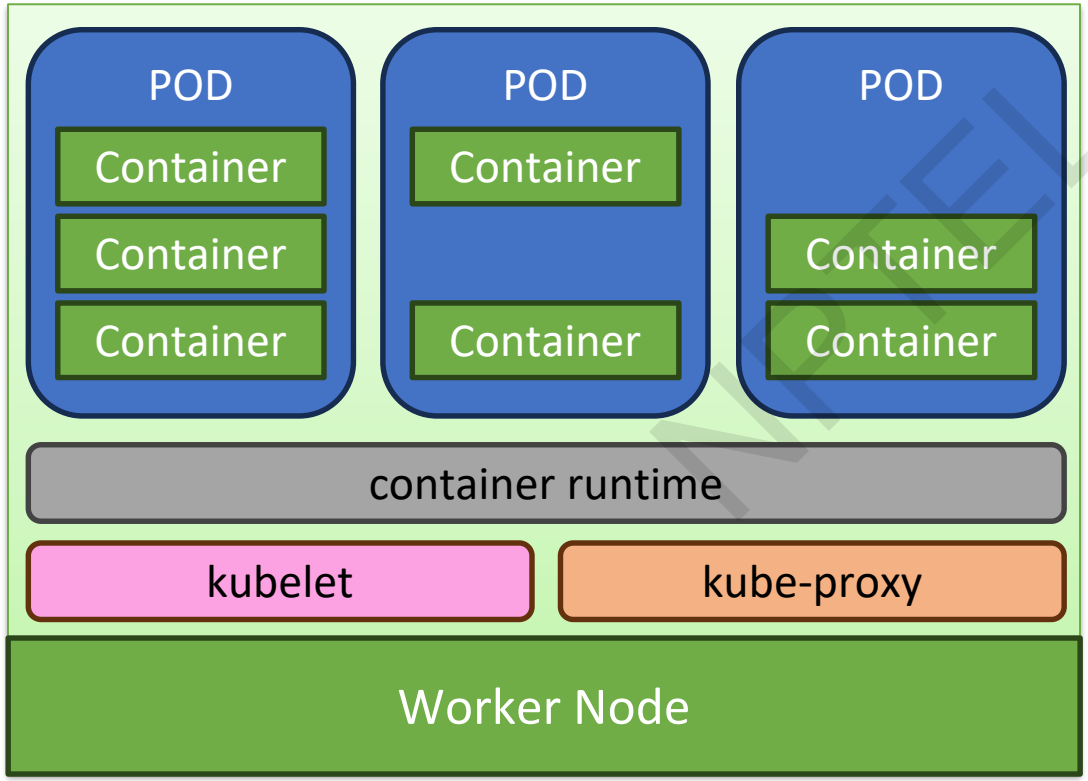
- **etcd** is a distributed key-value store. (a database)
- It is used by the API server and other components of the control plane to store and retrieve **information about the cluster.**
- It stores the cluster's persistent state which includes **object** data
- ***what are Kubernetes objects?*** *they* are entities to represent the **state of a cluster.** for e.g., they can describe:
    - What containerized applications are running (and on which nodes)
    - The resources available to those applications
    - The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance

- The **scheduler** is responsible for scheduling **pods** onto the **worker nodes** in the cluster.
- It uses information about the resources required by the pods and the available resources on the worker nodes to make container placement decisions.

Transcript Notes

- The **controller manager** is responsible for running **controllers** that manage the **state of the cluster.**
  - In Kubernetes, **controllers** are control loops that watch the state of a cluster and then make, or request changes where-ever needed.
  - It is similar to a *control loop* (like in robotics and automation) that is a non-terminating loop that regulates the state of a system.
  - Each controller tries to move the current cluster state closer to the **desired state.**
  - Controllers can also be created by users/engineers using the Kubernetes API.

Kubernetes Cluster Worker Nodes



Transcript Notes

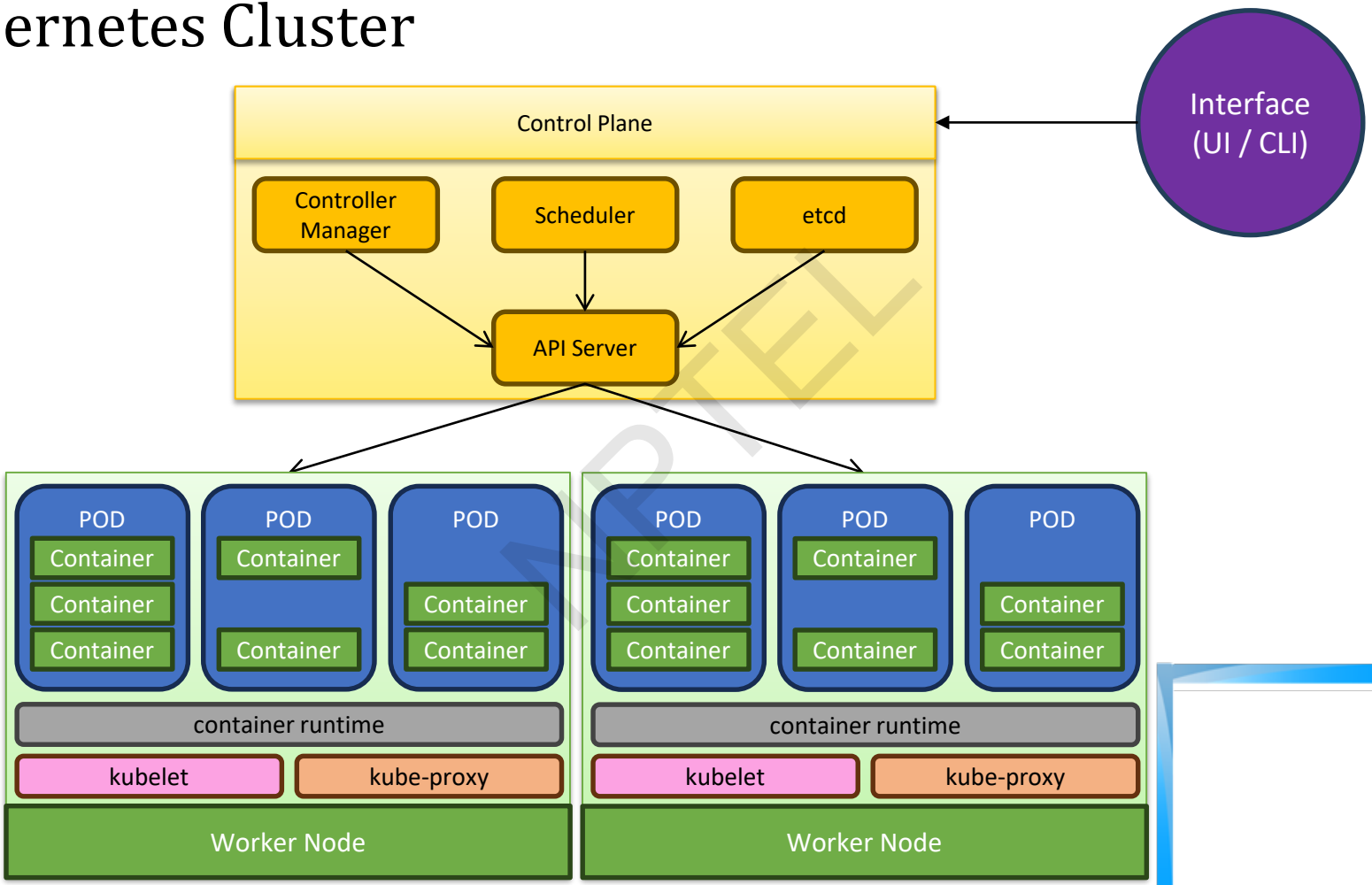
The core components of Kubernetes that run on the **worker nodes** include
kubelet, **container runtime**, and **kube-proxy**.

The **kubelet** is a daemon that runs on each worker node.
It is responsible for communicating with the control plane.
It receives instructions from the control plane about which pods to run on the node and ensures that the desired state of the pods is maintained.

The **container runtime** runs the containers on the worker nodes.
It is responsible for pulling the container images from a registry, starting and stopping the containers, and managing the containers' resources.

The **kube-proxy** is a network proxy that runs on each worker node. It is responsible for routing traffic to the correct pods.
It also provides load balancing for the pods and ensures that traffic is distributed evenly across the pods.

Kubernetes Cluster



Transcript Notes

- Lets us look into some more details,

- Regardless of the number of worker nodes in a Kubernetes cluster, they all become a **single place to deploy applications** using the **Kubernetes API**.
- Hence, instead of deploying an application to a server or a VM, we deploy it to a “cluster” instead.

- a very important component of Kubernetes which enables those nodes worker nodes and master nodes talk to each other is the virtual network that spans all the nodes that are part of the cluster
- in simple words virtual network turns all the nodes inside of a cluster into **one powerful machine** that can be seen as the sum of all the resources of individual nodes

Transcript Notes

- The **Kubernetes API Server** exposes a RESTful API. The client using the cluster and the other Kubernetes components can **create objects** via this API.
- The API server has the following functions:
    - Provides a consistent way of storing **objects** in **etcd**.
    - Performs validation of those **objects** so clients can't store improperly configured objects.
    - Provides an API to create, update, modify, or delete a **resource -** A **Kubernetes resource** is an endpoint in the **Kubernetes API** that stores a collection of **objects** of a certain kind; for example, the built-in “pods” resource-type contains a collection of Pod objects.
    - API server also Performs authentication and authorization of a request that the client sends.
    - and is Responsible for admission control if the request is trying to create, modify, or delete a resource.
- It should be noted that the only component that talks to etcd directly is the API server.
- All other components read and write data to etcd indirectly through the API server.
- etcd also implements a **watch feature**, which provides an event-based interface for asynchronously monitoring changes stored data (keys).
- when a key changes, its watchers get notified.
- The API server component relies on this notification to monitor and move the current state of **etcd** towards the **desired state.**
- In Kubernetes, **controllers** are components responsible for managing and maintaining a desired state for various resources.
- They help ensure that the current state of the system matches the **desired state.**

Transcript Notes

- Almost every **Kubernetes object** includes two nested object fields that govern the object's configuration: the object-***spec*** and the object-***status***.
- The **spec** describes the ***desired state*** of the object. (the spec field is set at the time of creating the object)
- The **status** describes the ***current state*** of the object, supplied and updated by the Kubernetes system and its components.
- In Kubernetes, **controllers** are components responsible for managing and maintaining a **desired state** for various resources.
- They help ensure that the current state of the system matches the **desired state**.

Transcript Notes

- a controller can track one or more Kubernetes resource type, examples of some Controllers :

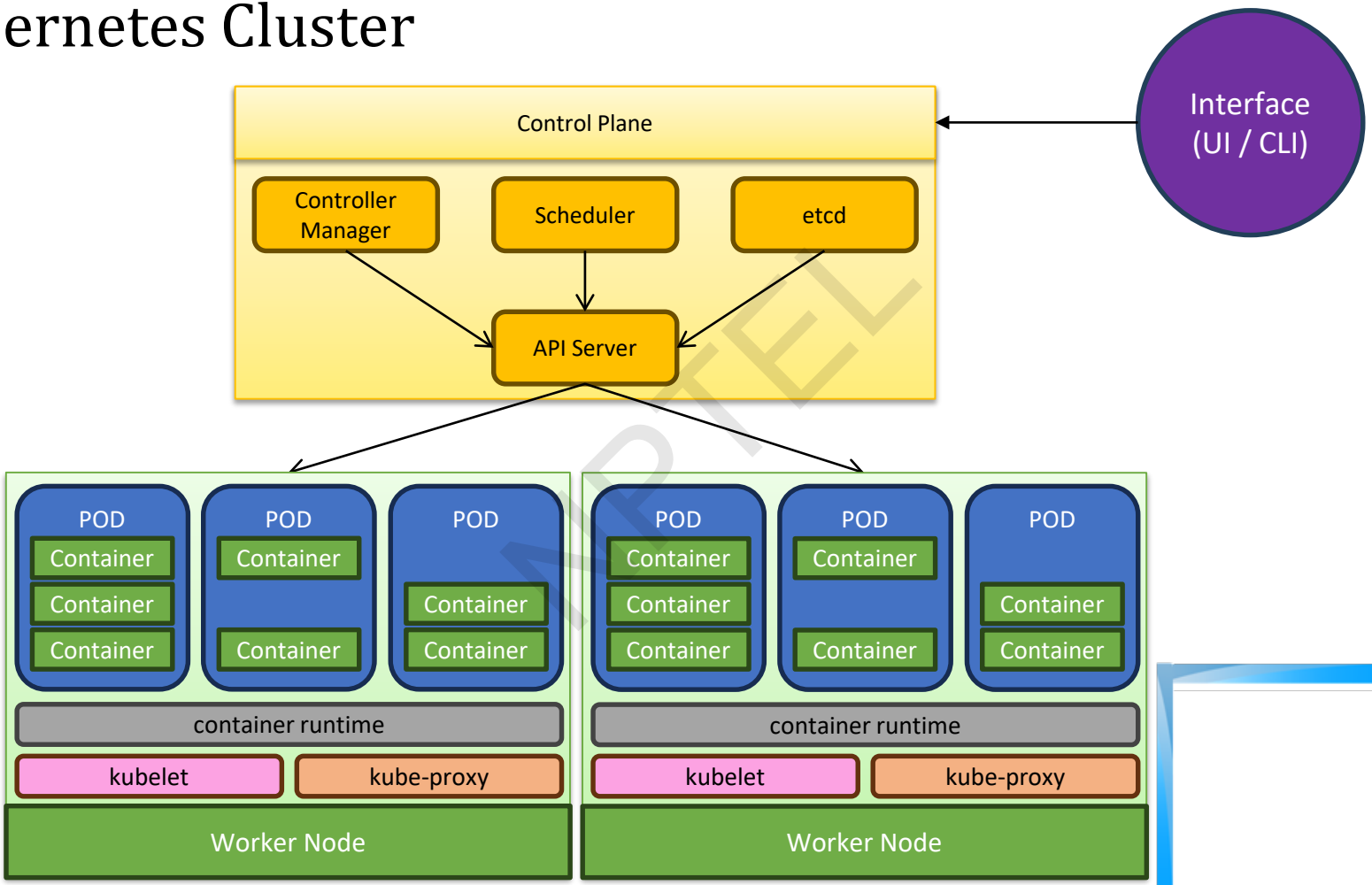
1. **ReplicaSet** Controller**:** It ensures a specified number of replicas of a pod are running at all times.
2. **Deployment** Controller**:** Deployments provide a declarative way to manage applications. They use ReplicaSets under the hood but offer additional features such as rolling updates and rollbacks. Deployments are commonly used for managing application lifecycle changes.
3. **StatefulSet** Controller**:** these are used to manage **stateful** applications, providing guarantees about the ordering and uniqueness of pods. They are useful for applications that require stable network identities and persistent storage.
4. **Namespace Controller:** In Kubernetes, a Namespace is a way to divide cluster resources between multiple users, teams, or projects. It provides a scope for names, ensuring that names of resources within a namespace are unique. The Namespace controller is responsible for managing these namespaces within a Kubernetes cluster.
5. **Node controller** - The Node controller is one of the core components in the Kubernetes control plane, and its primary responsibility is to manage the lifecycle of nodes within the cluster.
6. **Service Controller (Endpoint Controller):** The Service Controller is responsible for managing the lifecycle of Service resources in Kubernetes. When you create or modify a Service, the Service Controller ensures that the corresponding Endpoints are created or updated.
   what are services and Endpoints?
   1. **Service:** A Kubernetes Service provides a stable IP address and port for a set of Pods, allowing other applications within or outside the cluster to access them. The Service Controller manages the creation and maintenance of Services.
   2. **Endpoints:** The Endpoints resource in Kubernetes represents a set of addresses corresponding to the Pods backing a Service. When Pods are created or modified, the Service Controller updates the Endpoints to reflect the current set of healthy Pod addresses.

7. **DaemonSet:** DaemonSets ensure that all or some nodes run a copy of a pod. They are often used for cluster-level operations, such as running a monitoring agent on every node.
8. **Job and CronJob:** "Jobs" manage one or more pods to perform a particular task and ensure completion. "CronJobs" are similar

Transcript Notes

- Now, The **Scheduler** is responsible for assigning pods to nodes.
- The scheduling process works as follows,
    - The scheduler watches for newly created pods that have no nodes assigned.
    - For every pod that the Scheduler discovers, the Scheduler becomes responsible for finding the best node for that pod to run on.
    - Nodes that meet the scheduling requirements for a pod get called **feasible nodes**.
    - If none of the nodes are suitable, the pod remains unscheduled until the Scheduler can place it.
    - Once it finds a feasible node, it runs a set of functions to score the nodes, and the node with the highest score gets selected.
    - It then notifies the API server about the selected node. This is also called **process binding.**

Kubernetes Cluster



Transcript Notes

On the Workload Plane or Data Plane Side we have,

The **kubelet,** that is a daemon that runs on each worker node.
It is responsible for communicating with the control plane.
it talks to the API server and manages the applications running on its node.
It receives instructions from the control plane about which pods to run on the node and ensures that the desired state of the pods is maintained.
It reports the status of applications and the node via the API.

The main functions of kubelet service are:

- Register the node it's running on by creating a node resource in the API server.
- Continuously monitor the API server for pods that got scheduled to the node.
- Start the pod's containers by using the configured container runtime.
- Continuously monitor running containers and report their status, events, and resource consumption to the API server.
- Run the container liveness probes, restart containers when the probes fail and terminate containers when their pod gets deleted from the API server (notifying the server about the pod termination).

Transcript Notes

The **Container Runtime**, which can be Docker or any other runtime compatible with Kubernetes. It runs applications in containers as instructed by the Kubelet. There are two categories of container runtimes:

1. Lower-level container runtimes: These focus on running containers and setting up the *namespace* and *cgroups* for containers.

2. Higher-level container runtimes (container engine): These focus on formats, unpacking, management, sharing of images, and providing APIs for developers.

Container runtime can perform tasks such as:

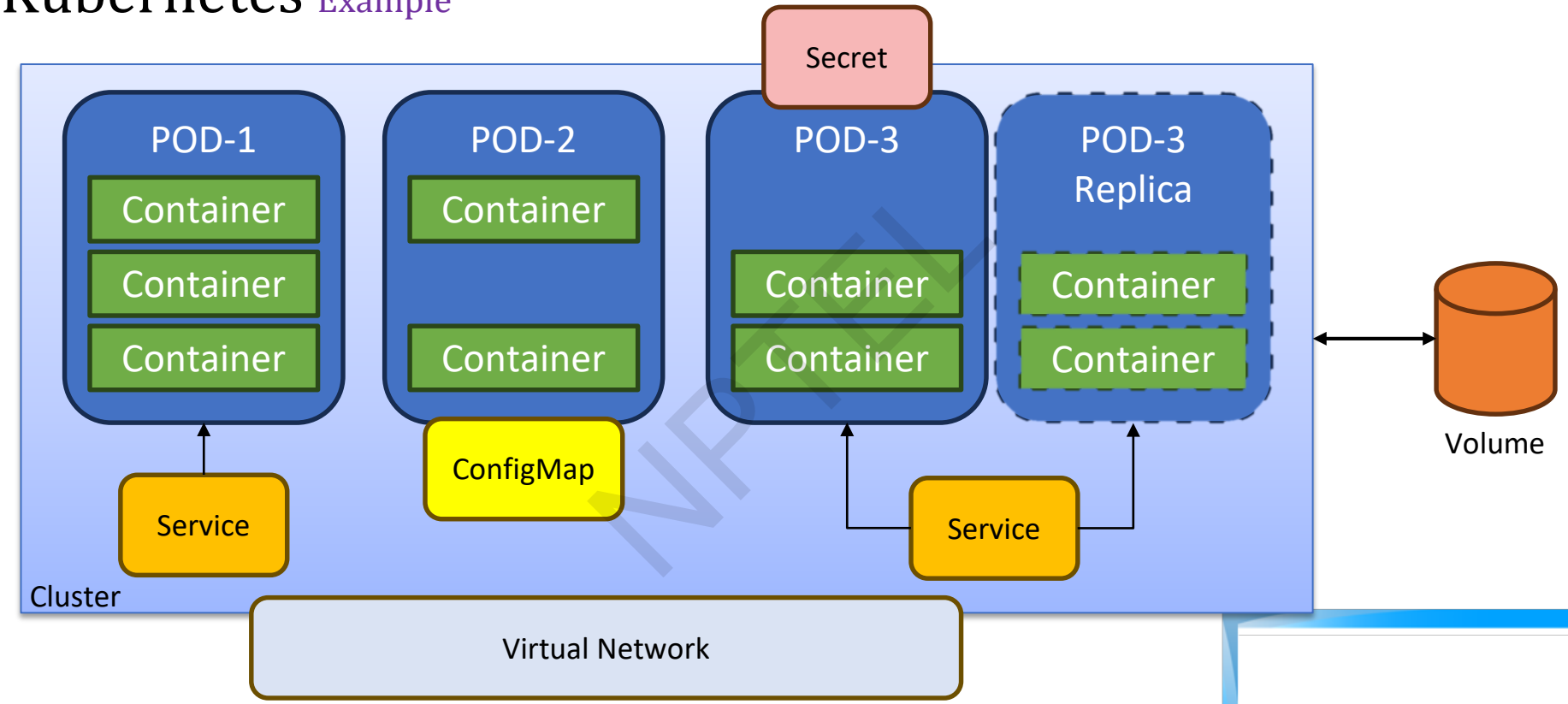
- Pulling the required container image from an image registry if it's not available locally.
- Prepares a container mount point.
- Alerts the kernel to assign some resource limits like CPU or memory limits.
- Pass system call (syscall) to the kernel to start the container.

3. The Kubernetes Service Proxy (Kube Proxy) load-balances network traffic between applications. The service proxy (kube-proxy) runs on each node and ensures that one pod can talk to another pod, one node can talk to another node, and one container can talk to another container.

It is responsible for watching the API server for changes on services and pod definitions to maintain that the entire network configuration is up to date.

When a service gets backed by more than one pod, the proxy performs load balancing across those pods.

Kubernetes Example



Transcript Notes

Now, let us look at some of the Kubernetes components from the view-point of an application deployment process.

Setup:

- suppose we have an application that requires multiple containers, each hosting a micro-service
- although we can run all the application-containers inside a single pod, it may be prone to node-failure since the pod will run on a single node
- instead, we can group applications to run in separate pods – that can be spread across multiple nodes
- In this case, the pods must be able to communicate with each other as per requirement of the application
- to facilitate communication between pods, Kubernetes offers a **Cluster network** which assigns an IP-address to each-pod
- **Cluster network is** A set of links, that facilitate communication within a cluster
- Note that this IP-address is assigned to each pod and NOT to the container
- each pod can communicate with each other using this ip address which is an internal ip address

- however, pods are **ephemeral** - which means that they can die very easily, for example, the application (or micro-service) running inside the container might crash.
- and when that happens, the pod will "die" and a new one will get created in its place
- consequently, it will be assigned a new ip address which may become inconvenient if our applications are directly using the IP address to communicate.
- To address this issue, another component of Kubernetes called a **"service"** is used

Transcript Notes

- The Service API, is an abstraction to expose groups of Pods over a network.
- Each Service object defines a logical set of endpoints (usually these endpoints are Pods) along with a policy about how to make those pods accessible.
- For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves. The Service abstraction enables this decoupling.
- it should be noted that lifecycles of service and the pod are not-connected
- this means that the service will stay the same even when the pod dies
- so, pods communicate with each other using a **service**

Transcript Notes

- A **ConfigMap** is an API object used to store non-confidential data in key-value pairs.
- Pods can consume ConfigMaps as *environment variables*, command-line arguments, or as configuration files stored in a volume (data storage).
- A ConfigMap allows to decouple **environment-specific** configuration from container images, so that your applications are easily portable.
- for e.g., we can specify the same application to run with a different command-line arguments in different containers or pods.

- A **Secret** is an object that contains a small amount of sensitive data such as a password, a token, or a key.
- Such information might otherwise be put in a Pod specification or in a container image.
- Using a Secret means that you don't need to include confidential data in your application code.
- Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods.
- Secrets are similar to ConfigMaps but are specifically intended to hold confidential data.
- Secrets can be used for various purposes such as the following:
    - Setting up **environment variables** for a container.
    - Providing credentials such as SSH keys or passwords to Pods.
    - Allow the kubelet to pull container images from private registries using login credentials
- The Kubernetes control plane also uses Secrets; for example, bootstrap token Secrets are a mechanism to help **automate node registration process**

Transcript Notes

- **Persistent Data-Storage in** Kubernetes:
- Kubernetes provides **Volumes** to persist the data generated by containers to the permanent storage.
- by default, containers have no way to store data persistently since all the data associated with the container is removed when the container stops running.
- Volumes abstracts an external storage and presents it to the containers running inside a pod so that data can be permanently stored

- **Replication:** Kubernetes allows a pod to be replicated. This is useful for 2 major reasons:
  - 1. Availability (fault-tolerance) - in case a node on the cluster fails
  - 2. Load-balancing – split the workload across multiple replicas of the same application
    - Note: a service is common to all the replicas of a pod. that is to say, replicated pods have the same network-address
- Usually, the deployment process of an application requires creating a "**Deployment-Configuration**" (called "**Deployment**" in Kubernetes)
- Given that an application may be composed of multiple pods, the replication factor (number of replicas) is usually defined for the **set of pods that the application uses.**
- The replication factor can be defined in the **Deployment-Configuration**
- **Deployment in** Kubernetes also allows for scaling of applications as well

Transcript Notes

- Some pods are stateful. for e.g., - a database application container. In such a case, we cannot use a **Deployment**, instead we should use **StatefulSet**.
- StatefulSet in Kubernetes manages the deployment and scaling of a set of Pods, *and provides guarantees about the ordering and uniqueness* of these Pods.
- Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec.
- Unlike a Deployment, a StatefulSet maintains a sticky identity for each of its Pods.
- These pods are created from the same spec, but are not interchangeable i.e, each pod has a persistent identifier that it maintains across any rescheduling.
- StatefulSets are useful for applications that require one or more of the following.
    - Stable, unique network identifiers.
    - Stable, persistent storage.
    - Ordered, graceful deployment and scaling.
    - Ordered, automated rolling updates.

- The concepts discussed so far, we can deploy applications on a Kubernetes cluster

Power of Kubernetes to deploy software on edge devices

- Architecture diagram shows works flow from the cloud through the virtual kubelet through the edge provider down to all of edge devices
- First, the virtual kubelet project lets you create a virtual node in your Kubernetes cluster, a virtual node is not a VM like most other nodes in the Kubernetes cluster instead it is an abstraction of a Kubernetes node that is provided by the virtual kubelet.
- Backing it, is an IOT hub, it can schedule workloads to it and treat it like any other Kubernetes node.

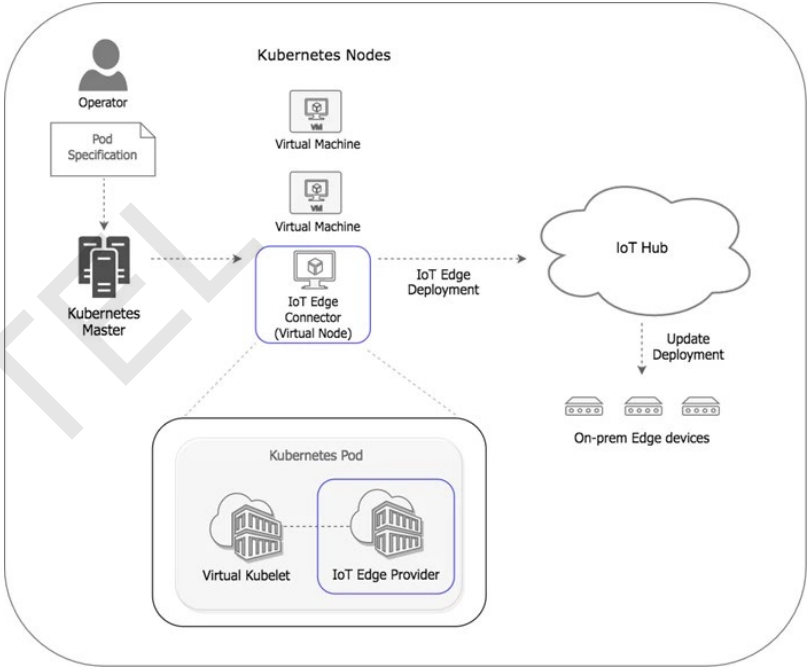


Image Source: https://azure.microsoft.com/es-es/blog/manage-azure-iot-edge-deployments-with-kubernetes/

Transcript Notes

- With a general overview of the components that make up Kubernetes we can understand how to deploy an application in Kubernetes.
- An application consists of several types of these objects - one type represents the application deployment as a whole, another represents a running instance of your application, another represents the service provided by a set of these instances and allows reaching them at a single IP address, and there are many others. These objects are usually defined in one or more manifest files in either YAML or JSON format.

These actions take place when we deploy an application on a Kubernetes cluster:

1. submit the application manifest to the Kubernetes API. The API Server writes the objects defined in the manifest to etcd.
2. A controller notices the newly created objects and creates several new objects - one for each application instance.
3. The Scheduler assigns a node to each instance.
4. The Kubelet notices that an instance is assigned to the Kubelet's node. It runs the application instance via the Container Runtime.
5. The Kube Proxy notices that the application instances are ready to accept connections from clients and configures a load balancer for them.
6. The Kubelets and the Controllers monitor the system and keep the applications running.

For, e.g, as shown in the figure – deploying applications in IoT cloud-Edge system

- Architecture diagram shows works flow from the cloud through the virtual kubelet through the edge provider down to all of your edge devices
- First, the virtual kubelet project lets you create a virtual node in your Kubernetes cluster, a virtual node is not a VM like most other nodes in the Kubernetes cluster instead it is an abstraction of a Kubernetes node that is provided by the virtual kubelet.
- Backing it, is an IOT hub, it can schedule workloads to it and treat it like any other Kubernetes node.

Power of Kubernetes to deploy software on edge devices

- When workloads are scheduled to this virtual node, edge provider comes in.
- The edge connector or the edge provider which are working in tandem with the virtual kubelet it takes the workload specification that comes in from Kubernetes and converts it into an IOT edge deployment.
- Then the IOT edge deployment is shipped back to the backing IOT hub for this virtual node.
- Lastly, the IOT hub in turn pushes this deployment down to all the targeted devices.

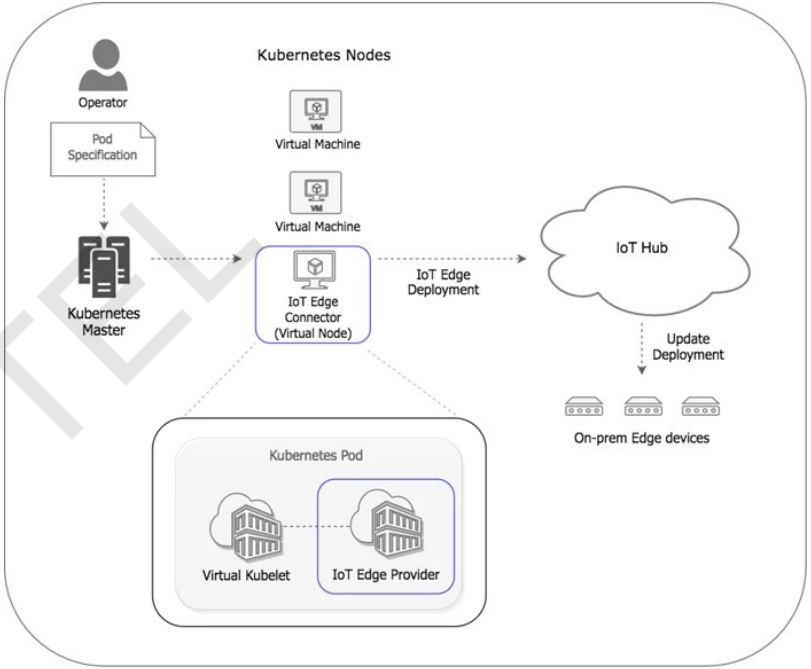


Image Source: https://azure.microsoft.com/es-es/blog/manage-azure-iot-edge-deployments-with-kubernetes/

Transcript Notes

- When workloads are scheduled to this virtual node, edge provider comes-in, as depicted.
- The edge connector or the edge provider which are working in tandem with the virtual kubelet it takes the workload specification that comes in from Kubernetes and converts it into an IOT edge deployment.
- Then the IOT edge deployment is shipped back to the backing IOT hub for this virtual node.
- Lastly, the IOT hub in turn pushes this deployment down to all the targeted devices.

Benefits of using Kubernetes

- Self-service deployment of applications
- Reducing costs via better infrastructure utilization
- Automatically adjusting to changing load
- Keeping applications running smoothly
- Simplifying application development

Transcript Notes

The benefits of using Kubernetes You've already learned why many organizations across the world have welcomed Kubernetes into their data centers. Now, let's take a closer look at the specific benefits it brings to both development and IT operations teams.

Self-service deployment of applications
Because Kubernetes presents all its worker nodes as a single deployment surface, it no longer matters which node you deploy your application to. This means that developers can now deploy applications on their own, even if they don't know anything about the number of nodes or the characteristics of each node. In the past, the system administrators were the ones who decided where each application should be placed. This task is now left to Kubernetes. This allows a developer to deploy applications without having to rely on other people to do so. When a developer deploys an application, Kubernetes chooses the best node on which to run the application based on the resource requirements of the application and the resources available on each node.

Reducing costs via better infrastructure utilization If you don't care which node your application lands on, it also means that it can be moved to any other node at any time without you having to worry about it. Kubernetes may need to do this to make room for a larger application that someone wants to deploy. This ability to move applications allows the applications to be packed tightly together so that the resources of the nodes can be utilized in the best possible way.

Automatically adjusting to changing load
Using Kubernetes to manage your deployed applications also means that the operations team doesn't have to constantly monitor the load of each application to respond to sudden load peaks. Kubernetes takes care of this also. It can monitor the resources consumed by each application and other metrics and adjust the number of running instances of each application to cope with increased load or resource usage. When you run Kubernetes on cloud infrastructure, it can even increase the size of your cluster by provisioning additional nodes through the cloud provider's API. This way, you never run out of space to run additional instances of your applications.

Transcript Notes

Keeping applications running smoothly

Kubernetes also makes every effort to ensure that your applications run smoothly. If your application crashes, Kubernetes will restart it automatically. So even if you have a broken application that runs out of memory after running for more than a few hours, Kubernetes will ensure that your application continues to provide the service to its users by automatically restarting it in this case. Kubernetes is a self-healing system in that it deals with software errors like the one just described, but it also handles hardware failures. As clusters grow in size, the frequency of node failure also increases. For example, in a cluster with one hundred nodes and a MTBF (mean-time-between-failure) of 100 days for each node, you can expect one node to fail every day. When a node fails, Kubernetes automatically moves applications to the remaining healthy nodes. The operations team no longer needs to manually move the application and can instead focus on repairing the node itself and returning it to the pool of available hardware resources. If your infrastructure has enough free resources to allow normal system operation without the failed node, the operations team doesn't even have to react immediately to the failure. If it occurs in the middle of the night, no one from the operations team even has to wake up. They can sleep peacefully and deal with the failed node during regular working hours.

Simplifying application development

The improvements described in the previous section mainly concern application deployment. But what about the process of application development? Does Kubernetes bring anything to their table? It definitely does. As mentioned previously, Kubernetes offers infrastructure-related services that would otherwise have to be implemented in your applications. This includes the discovery of services and/or peers in a distributed application, leader election, centralized application configuration and others. Kubernetes provides this while keeping the application Kubernetes-agnostic, but when required, applications can also query the Kubernetes API to obtain detailed information about their environment. They can also use the API to change the environment.

Conclusion

- In this lecture we discussed about,
- Overview of Kubernetes – a container orchestration tool
- The need for using Kubernetes and its benefits.
- Components of Kubernetes – Control plane and Worker nodes
- Deployment of applications using Kubernetes

Thank You!

References

- *Container Networking, From Docker to Kubernetes - Michael Hausenblas, O'Reilly Media (2018)*
- *Ian Miell, Aiden Hobson Sayers - Docker in Practice-Manning Publications (2019)*
- *Kubernetes in Action, Manning Publications, Marko Lukša*
- *https://kubernetes.io/docs/concepts/architecture/*
- *https://kubernetes.io/docs/home/*

Week 3 Lecture 2

0 mins

NoSQL Databases and Key-Value Stores

Dr. Rajiv Misra, Professor
Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Contents

- Relational V/S NoSQL Databases
- Design and insight of NoSQL and Key-Value stores for today's Edge storage systems.
- NoSQL storage system - **Apache Cassandra**
- CAP Theorem and Consistency solutions.

Review: Relational Database

In a traditional Relation Database:

- Data stored in tables
- Schema is imposed – Structed Data
- Each row (data item) in a table has a unique primary key
- Queried using **SQL (Structured Query Language)**
- Supports joins
- Provides A.C.I.D properties
- Backed by BTree and B+Tree data structures – optimized for reads
- Scales well vertically (Scale up)

Relational Database Example

users table

user_id	name	zipcode	blog_url	blog_id
110	Smith	98765	smith.com	11
331	Antony	54321	antony.in	12
767	John	75676	john.net	13

Primary keys

Foreign keys

blog table

Id	url	last_updated	num_posts
11	smith.com	9/7/17	991
13	john.net	4/2/18	57
12	antony.in	15/6/16	1090

Example SQL queries

1. SELECT zipcode
   FROM users
   WHERE name = “John”

2. SELECT url
   FROM blog
   WHERE id = 11

3. SELECT users.zipcode, blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

Edge Workloads

- **Mismatch:**
  - Unstructured Data: Difficult to come with schemas where the data can fit
  - Lots of random reads and writes: Coming from clients distributed over the network
  - Foreign keys rarely needed & Joins are infrequent
  - Data caching requirements
  - Low Latency Read/Write
  - More Write Intensive

- **Needs of Today's Edge Workloads**
  - Speed – contributes towards reducing latency
  - Resources (compute, storage) distributed across nodes
  - Avoid Single point of Failure (SPoF)
  - Low TCO (Total cost of operation and Total cost of ownership)
  - Incremental Scalability
  - Horizontal Scalability

Scale out, not Scale up

- **Scale up (Vertical Scaling) = Increase Capacity (per head)**
  - Not cost-effective
  - Need to replace/upgrade machines often

- **Scale out (Horizontal Scaling) = Increase Quantity (no. of heads)**
  - Cost-effective
  - Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
  - Commonly used in datacenters and clouds today
  - Suitable for edge computing scenarios

NoSQL Data-Model

- NoSQL or *Not Only* SQL is a flexible Database Management Approach
- Supports variety of data-models:
  - Key-Value
  - Document
  - Tabular/Column
  - Graph
  - Multi-Model
- Designed for:
  - Large Amounts of data – high throughput
  - Distributed/Clustered Environment
  - Scalability
  - Partition Tolerance
  - High Availability (replication)

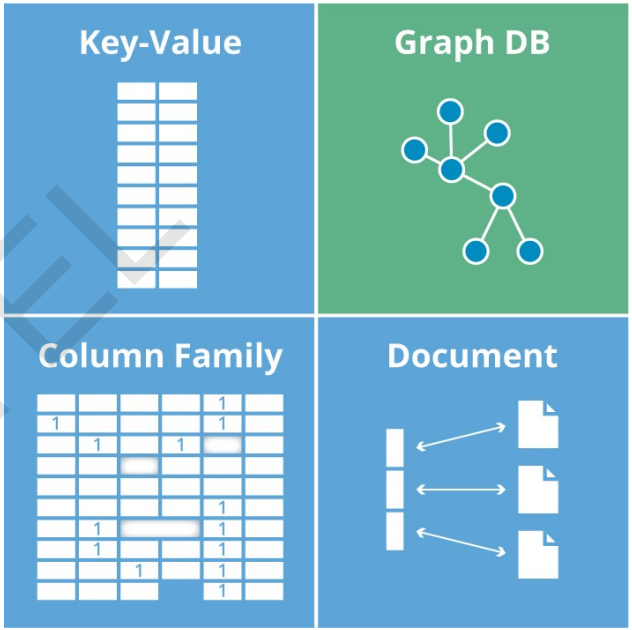


Figure Source: https://neo4j.com/blog/why-nosql-databases/

NoSQL Databases

Figure Source: https://www.decipherzone.com/blog-detail/nosql-databases

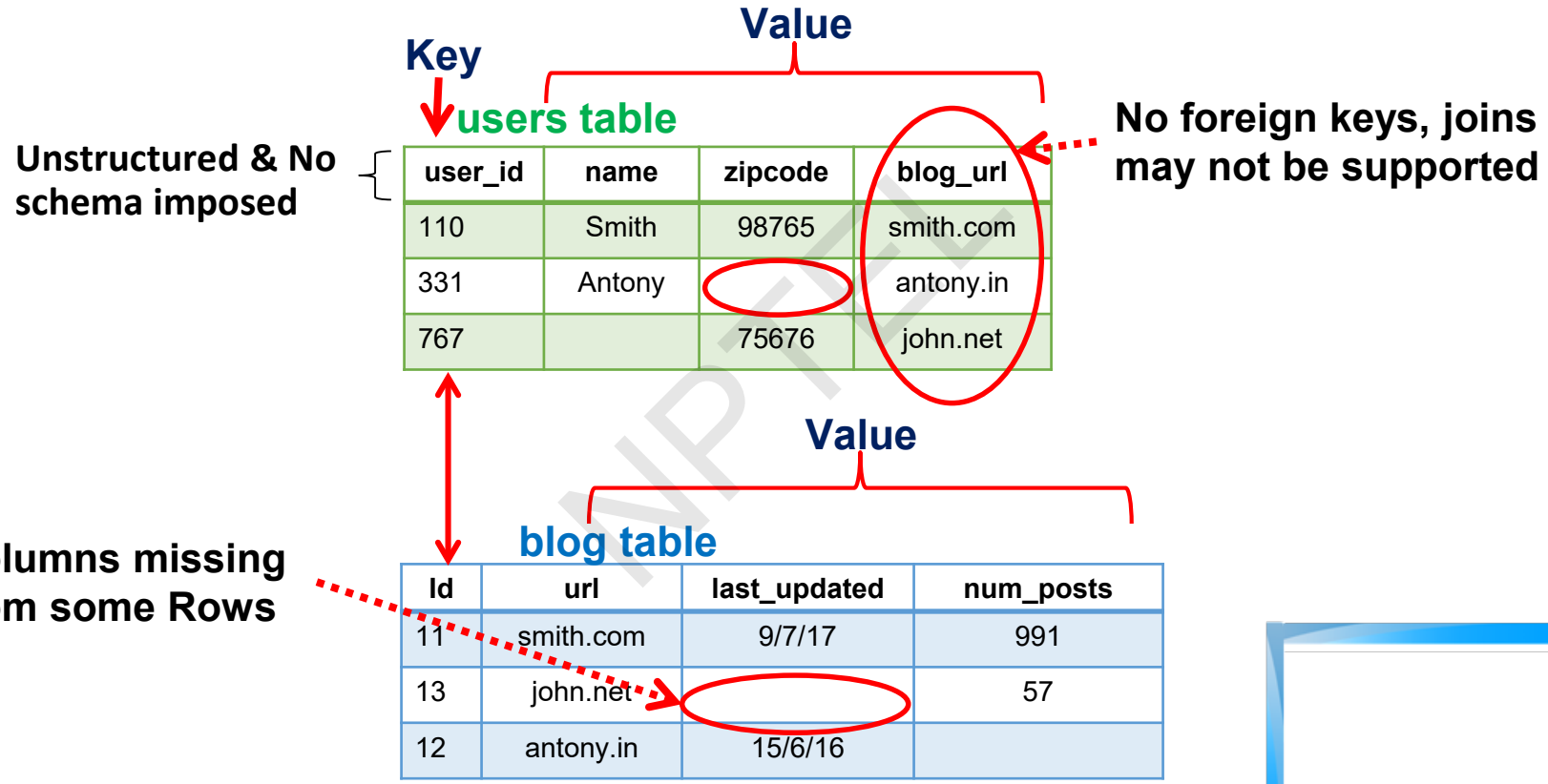
Key-Value Stores

- **A dictionary like data structure**
  - **C**reate-**R**ead-**U**pdate-**D**elete by **key**

- Each piece of data or **value** in the store is associated with a unique identifier known as a **key**.
- **Example:**
- **(Business) Key → Value**
- **(flipkart.com) item number → information about it**
- **(easemytrip.com) Flight number → information about flight, e.g., availability**
- **(twitter.com) tweet id → information about tweet**
- **(mybank.com) Account number → information about it**

A **key-value store** is a type of **database** where data is stored as **key-value** pairs.

Key-Value Data Model



Column-Oriented Storage

NoSQL systems often use column-oriented storage

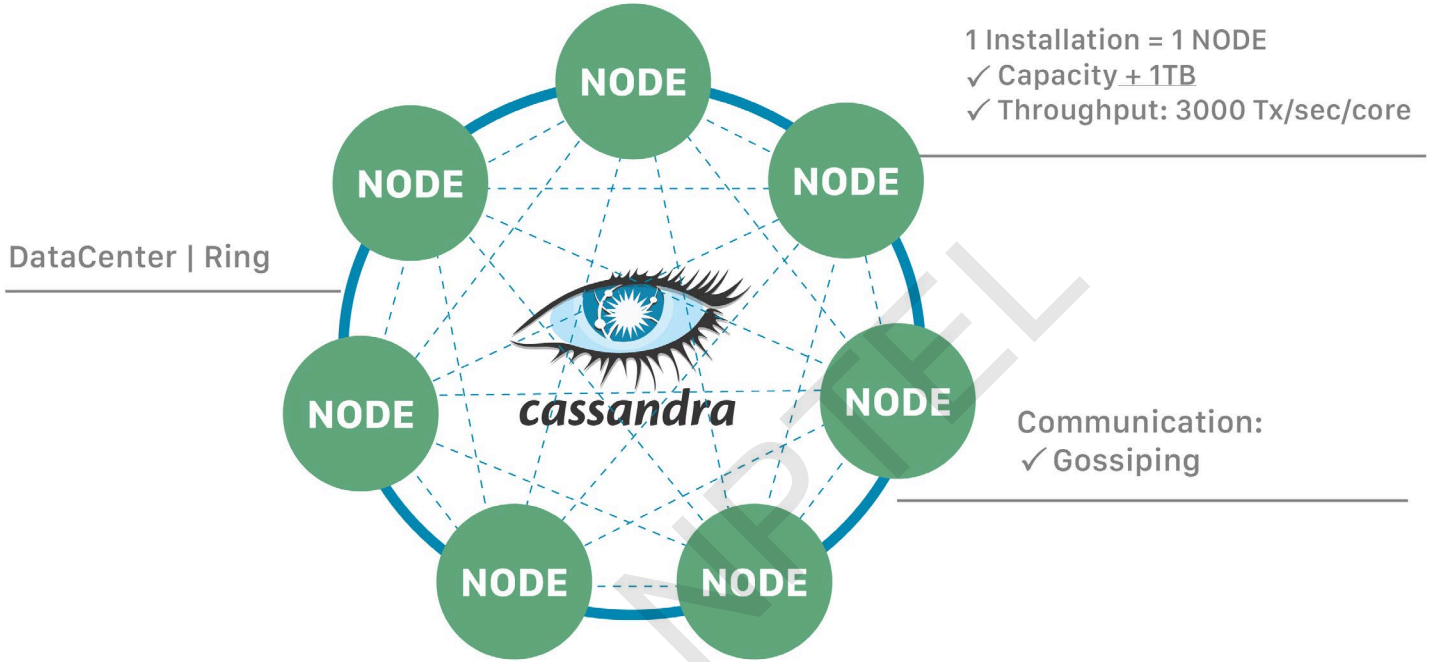
- Relation Databases store an entire row together (on disk or at a server)
- NoSQL systems typically store a column together (or a group of columns).
  - Entries within a column are indexed and easy to locate, given a key (and vice-versa)

Why useful?

- Range searches within a column are fast
  no need to fetch the entire rows
- E.g., Get me all the **blog_ids** from the blog table that were updated within the past month
- Search in the the **last_updated** column, fetch corresponding **blog_id** column
- Don't need to fetch the other columns

Design of Apache Cassandra

ApacheCassandra™= NoSQL Distributed Database



- Intended to run in a datacenter (and also across DCs)
- Originally designed at Facebook, Open-sourced later, today an Apache project
- Some of the companies that use Cassandra in their production clusters - IBM, Adobe, HP, eBay, Ericsson, Twitter, Netflix

Source: https://cassandra.apache.org/_/cassandra-basics.html

Cassandra Features

- **Distributed and Fault Tolerant:**
  - Can run on multiple machines while appearing to users as a unified whole
  - Allows **replication** across distributed nodes and online replacement of failed nodes.
  - No single point failure.
  - Nodes communicate with one another through a protocol called *gossip*, which is a process of computer peer-to-peer communication.
  - Masterless architecture – any node in the database can provide the exact same functionality as any other node – contributing to robustness and resilience.
  - Multiple nodes can be organized logically into a cluster, or "ring".

- **Elastic and Scalable**
  - Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.
  - can stream data between nodes during scaling operation.
  - can add new nodes during peak traffic.

Cassandra Data-Model

Provides **Cassandra Query Language** (CQL), an SQL-like language, to create, modify, and delete database schema, as well as access data. CQL allows users to organize data within a cluster of Cassandra nodes using:

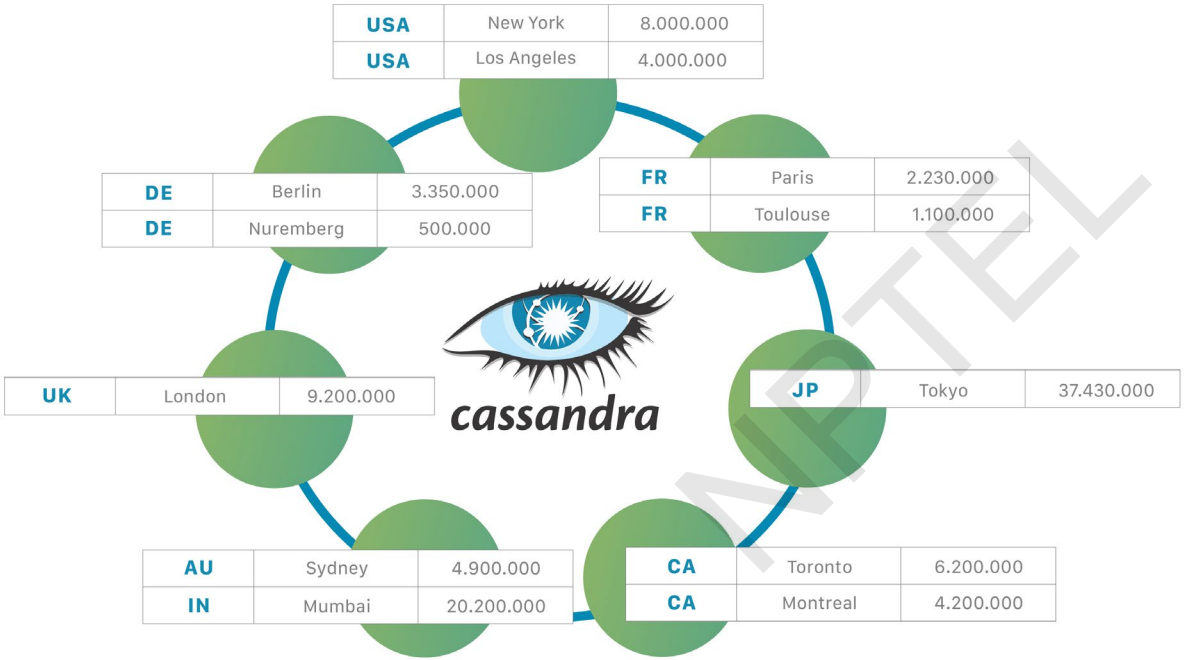
- **Keyspace**: contain tables.
- **Table**: Tables are composed of rows and columns. Tables are partitioned based on the columns provided in the partition key.
- **Partition**: Defines the mandatory part of the primary key all rows in Cassandra must have to identify the node in a cluster where the row is stored. All performant queries supply the partition key in the query.
- **Row**: Contains a collection of columns identified by a unique primary key made up of the partition key and optionally additional clustering keys.
- **Column**: A single datum with a type which belongs to a row. Columns define the typed schema for a single datum in a table.

Source: https://cassandra.apache.org/doc/4.1/cassandra/architecture/overview.html

Transcript Notes

- Partitions data over storage nodes using a special form of hashing called **consistent hashing**.
- Each **partition** can be **replicated** to multiple racks, physical nodes and even datacenters
  .
- Replication factor (RF) indicates number of copies of the partition that should exist.

Cassandra Data Partitioning



COUNTRY	CITY	POPULATION
USA	New York	8.000.000
USA	Los Angeles	4.000.000
FR	Paris	2.230.000
DE	Berlin	3.350.000
UK	London	9.200.000
AU	Sydney	4.900.000
DE	Nuremberg	500.000
CA	Toronto	6.200.000
CA	Montreal	4.200.000
FR	Toulouse	1.100.000
JP	Tokyo	37.430.000
IN	Mumbai	20.200.000

Partition Key

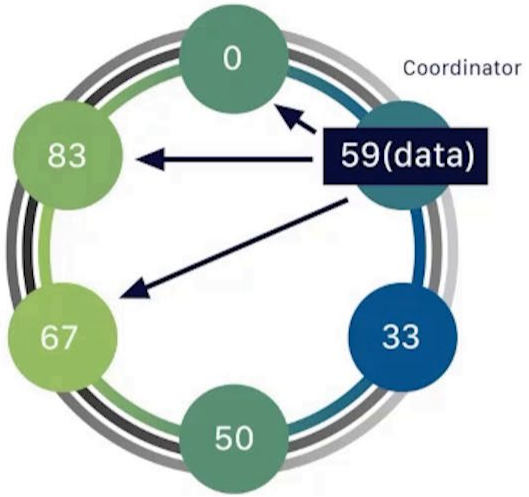
Source: https://cassandra.apache.org/_/cassandra-basics.html

Cassandra Data Partitioning

- When a mutation occurs, the coordinator **hashes** the **partition key** to determine the *token* range the data belongs to and then replicates the mutation to the replicas of that data according to the **replication strategy**.

- All replication strategies have the notion of a replication factor (RF) which indicates number of copies of the partition that should exist.

RF = 3



Source: https://cassandra.apache.org/_/cassandra-basics.html

Cassandra Data Partitioning

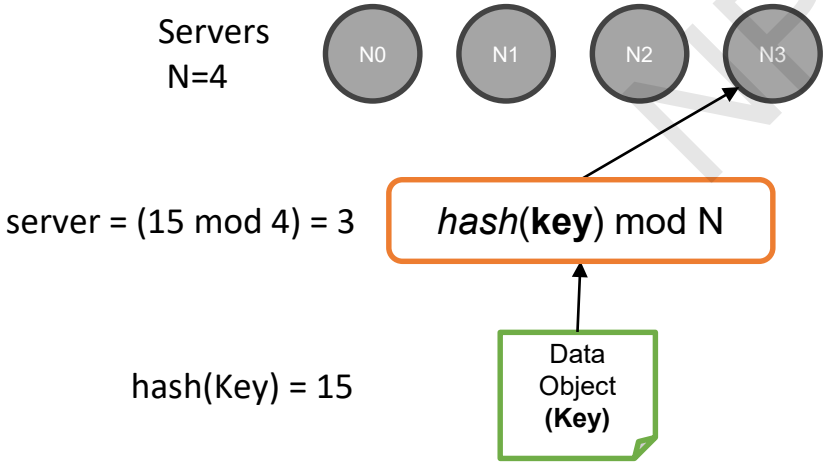
- Cassandra uses a **Last-Write-Wins model** to resolve conflicting mutations on replica sets.
- In a **last write wins** model, every mutation is timestamped (including deletes) and then the latest version of data is the "winning" value.
- **Self-healing**: when we have multiple replicas for the same data and if a node goes down, a hard drive fails, or AWS resets an instance. Replication ensures that data isn't lost.

Source: https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

Cassandra Consistent Hashing

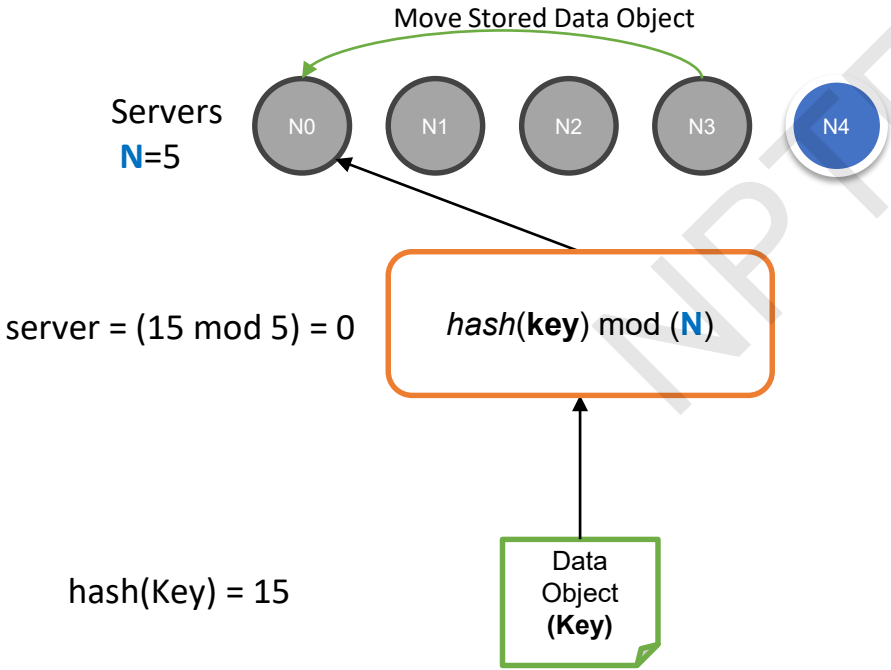
Naive Hashing v/s Consistent Hashing:

- **Naive Hashing using N buckets**
  - assign every server to a bucket between 0 and N
  - hash of (input key modulo N) → store the data on the associated server



Cassandra Consistent Hashing

- **Issue:** **adding or removing a single node require shifting almost all of the stored keys**

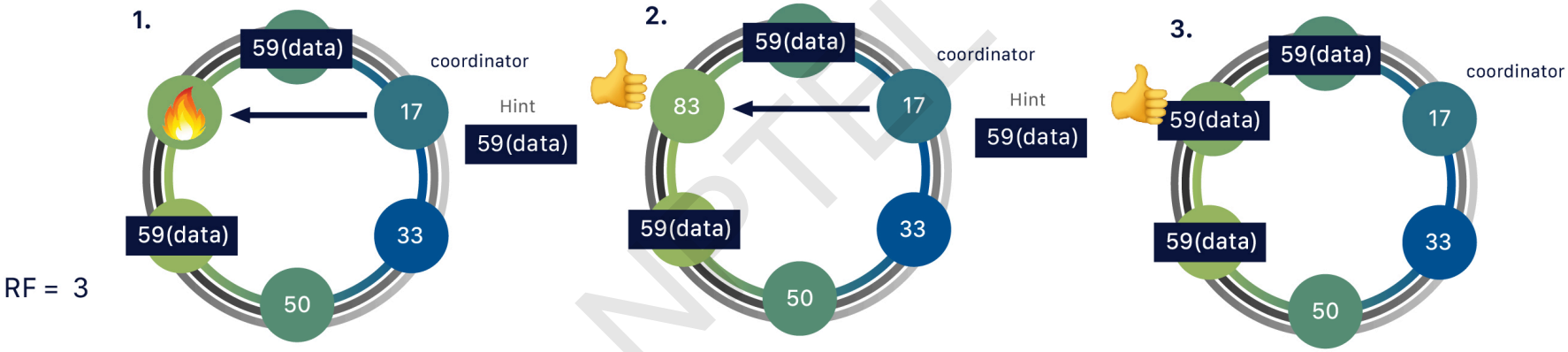


Suppose we have 16 objects (keys) stored on 4 servers distributed equally, with hash(key) ranging from 0 to 15, How many objects must be moved if we add one more server?

> 12 objects. All objects from keys 4 to 15 will have to be moved

Cassandra Data Partitioning

- **Self-healing**: when we have multiple replicas for the same data and if a node goes down, a hard drive fails, or AWS resets an instance. Replication ensures that data isn't lost.

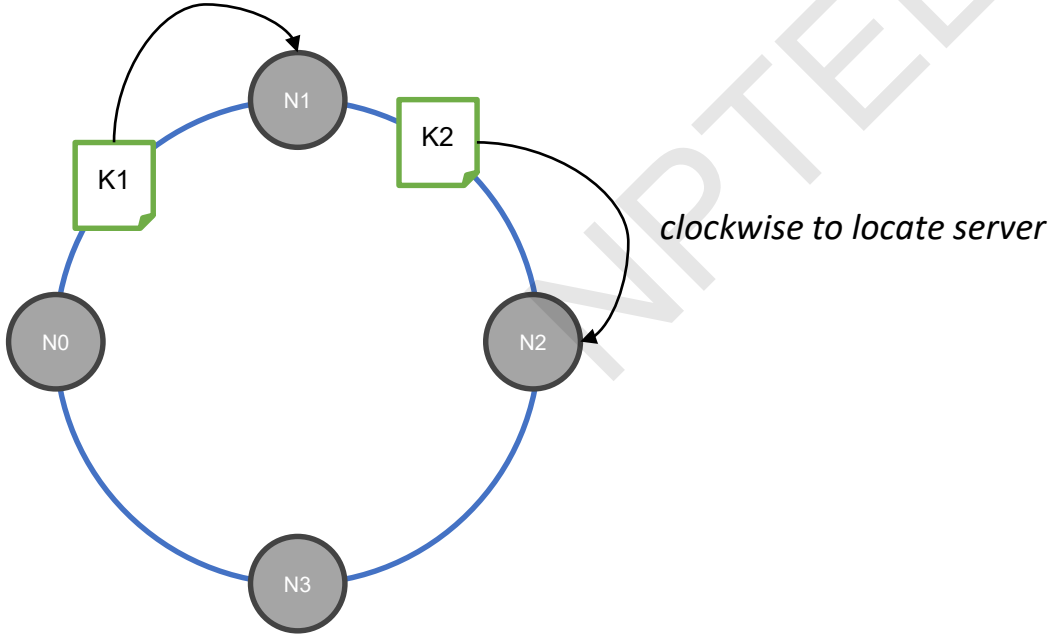


- Hinted Handoff: The coordinator stores a "hint" for that data as well, and when the downed replica comes back up, it will find out what it missed, and catch up to speed with the other two replicas. This is done completely automatically.

Source: https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

Cassandra Consistent Hashing

- Hash objects and servers using same hashing function to a common hash space.



Transcript Notes

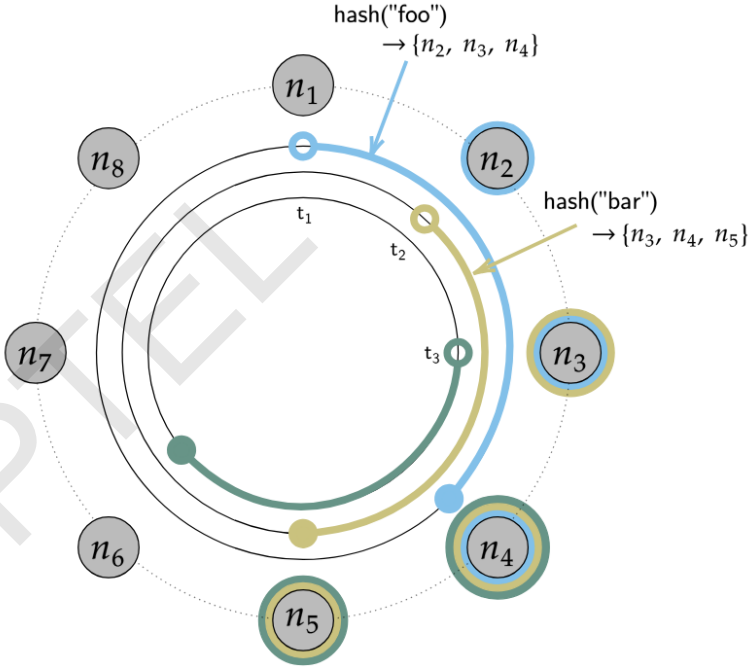
- Cassandra maps every node to one or more tokens on a continuous hash ring, and defines ownership by hashing a key onto the ring and then "walking" the ring in one direction, similar to the Chord algorithm. The main difference of consistent hashing to naive data hashing is that when the number of nodes (buckets) to hash into changes, consistent hashing only has to move a small fraction of the keys.

- Objects and servers are hashed using a common hash space
- To locate the server corresponding to a key, move in clockwise direction on the ring until first node is encountered

Cassandra Consistent Hashing

Hashing with replication: For example, if we have an eight-node cluster with evenly spaced tokens, and a **replication factor** (RF) of 3, then to find the owning nodes for a key:

- coordinator hashes that key to generate a token
- then "walk" the ring in a *clockwise* fashion until we encounter three distinct nodes, at which point we have found all the replicas of that key.



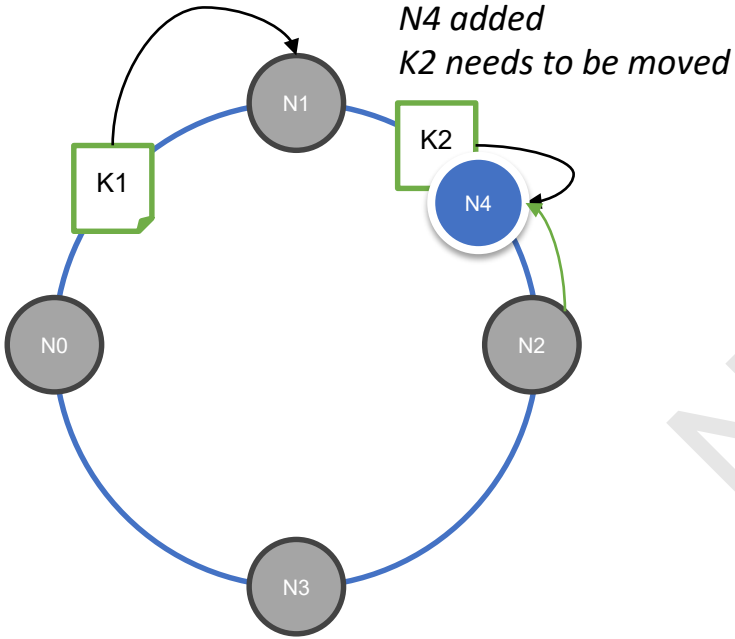
Source: https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

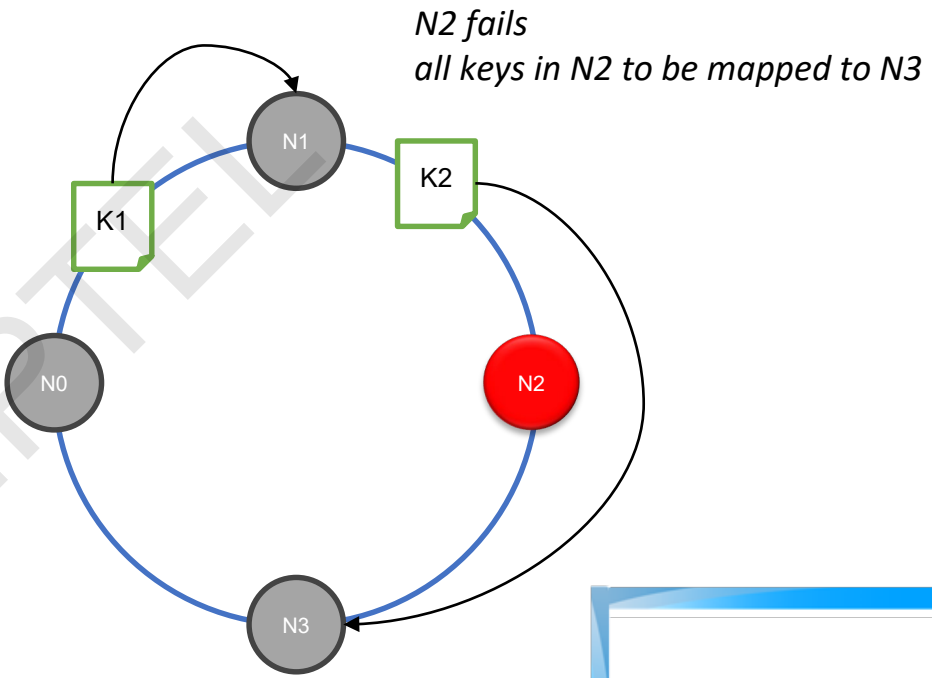
Transcript Notes

- **SOURCE:** https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

- For example, if we have an eight node cluster with evenly spaced tokens, and a replication factor (RF) of 3, then to find the owning nodes for a key we first hash that key to generate a token (which is just the hash of the key), and then we "walk" the ring in a clockwise fashion until we encounter three distinct nodes, at which point we have found all the replicas of that key. This example of an eight node cluster with RF=3 can be visualized as shown in the figure

Cassandra Consistent Hashing





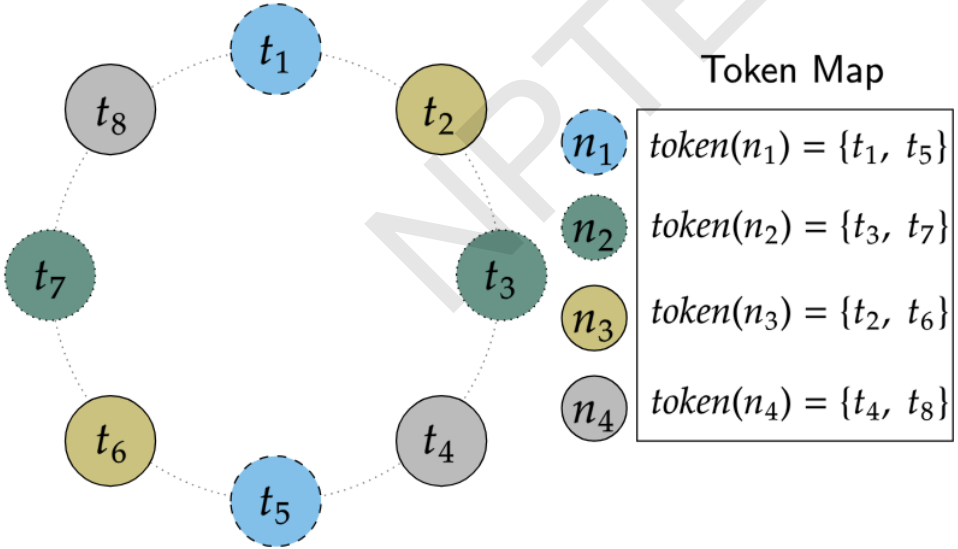
Transcript Notes

- In this case when adding new nodes, only a fraction of keys need to be moved to the newly added node
- In case a node on the ring fails, it requires to move all keys to the next node (clockwise) in the ring. it requires moving a significant fraction of keys in this case

Cassandra Virtual Nodes

Virtual Nodes in Cassandra - defines a mapping of **Tokens** (a single position on the ring) to **Endpoints** (A single server – physical IP) called the **Token** Map. Cassandra keeps track of what ring positions map to which physical endpoints.

For example, in the following figure we can represent an eight-node cluster using only four physical nodes by assigning two tokens to every node:



Source: https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

Transcript Notes

- **SOURCE**: https://cassandra.apache.org/doc/4.1/cassandra/architecture/dynamo.html

- CASSANDRA advocates for the use of "virtual nodes" to solve this imbalance problem.
- Cassandra introduces some nomenclature to handle VIRTUAL NODE concepts:
- **Token**: A single position on the hash ring.
- **Endpoint**: A single physical IP and port on the network.
- **Host ID**: A unique identifier for a single "physical" node, usually present at one **Endpoint** and containing one or more **Tokens**.
- **Virtual Node** (or **vnode**): A **Token** on the hash ring owned by the same physical node, one with the same **Host** ID.

- Virtual nodes solve the problem by assigning multiple tokens in the token ring to each physical node. By allowing a single physical node to take multiple positions in the ring, we can make small clusters look larger and therefore even with a single physical node addition we can make it look like we added many more nodes, effectively taking many smaller pieces of data from more ring neighbors when we add even a single node.

- For example, in the figure we can represent an eight-node cluster using only four physical nodes by assigning two tokens to every node:
- *Trade off: a greater number of Virtual Nodes also require more storage for meta-data*

Transcript Notes

- **Multiple tokens per physical node provide the following benefits:**

1. When a new node is added it accepts approximately equal amounts of data from other nodes in the ring, resulting in equal distribution of data across the cluster.
2. When a node is decommissioned, it loses data roughly equally to other members of the ring, again keeping equal distribution of data across the cluster.
3. If a node becomes unavailable, query load (especially token aware query load), is evenly distributed across many other nodes.

- Multiple tokens, however, can also have **disadvantages**:

1. Every token introduces up to 2 * (RF - 1) additional neighbors on the token ring, which means that there are more combinations of node failures where we lose availability for a portion of the token ring. The more tokens you have, the higher the probability of an outage.
2. Cluster-wide maintenance operations are often slowed. For example, as the number of tokens per node is increased, the number of discrete repair operations the cluster must do also increases.
3. Performance of operations that span token ranges could be affected.

Cassandra Data Replication

Every keyspace in Cassandra has its own replication strategy.

SimpleStatergy

- Allows a single integer Replication Factor (**RF**) to be defined which determines the number of nodes that should contain a copy of each row.
- Treats all nodes identically, ignoring any configured datacenters or racks.
- To determine the replicas for a token range,
  - Iterates through the tokens in the ring, starting with the token range of interest.
  - For each token, it checks whether the owning node has been added to the set of replicas and adds it if not present
  - This process continues until '**RF**' number of distinct nodes have been added to the set of replicas
- Useful only for testing clusters where the layout of the cluster (in the datacenter) is unknown

Cassandra Data Replication

Every keyspace in Cassandra has its own replication strategy.

`NetworkTopologyStrategy`

- requires a specified replication factor for each datacenter in the cluster.
- **Rack-Awareness**: attempts to choose replicas within a datacenter from different racks as specified by the **Snitch**
- If the number of racks is greater than or equal to the replication factor for the datacenter, each replica is guaranteed to be chosen from a different rack. Otherwise, each rack will hold at least one replica, but some racks may hold more than one.
- Useful for production environments with known layout of racks within datacenters.

Cassandra Snitches

- **Maps:** IPs to racks and DCs. Configured in cassandra.yaml config file
  - **SimpleSnitch**
    Unaware of Topology (Rack-unaware)
  - **RackInferring**
    Assumes topology of network by octet of server's IP address e.g., 101.102.103.104 = x.<DC octet>.<rack octet>.<node octet>
  - **PropertyFileSnitch:** uses a config file
  - **EC2Snitch:** uses EC2.
    - EC2 Region = DC
    - Availability zone = rack
- Other snitch options available

Transcript Notes

A snitch determines which datacenters and racks nodes belong to. They inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into datacenters and racks. Specifically, the replication strategy places the replicas based on the information provided by the new snitch. All nodes must return to the same rack and datacenter. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).

SimpleSnitch
The SimpleSnitch is used only for single-datacenter deployments. It does not recognize datacenter or rack information and can be used only for single-datacenter deployments or single-zone in public clouds. It treats strategy order as proximity, which can improve cache locality when disabling read repair.

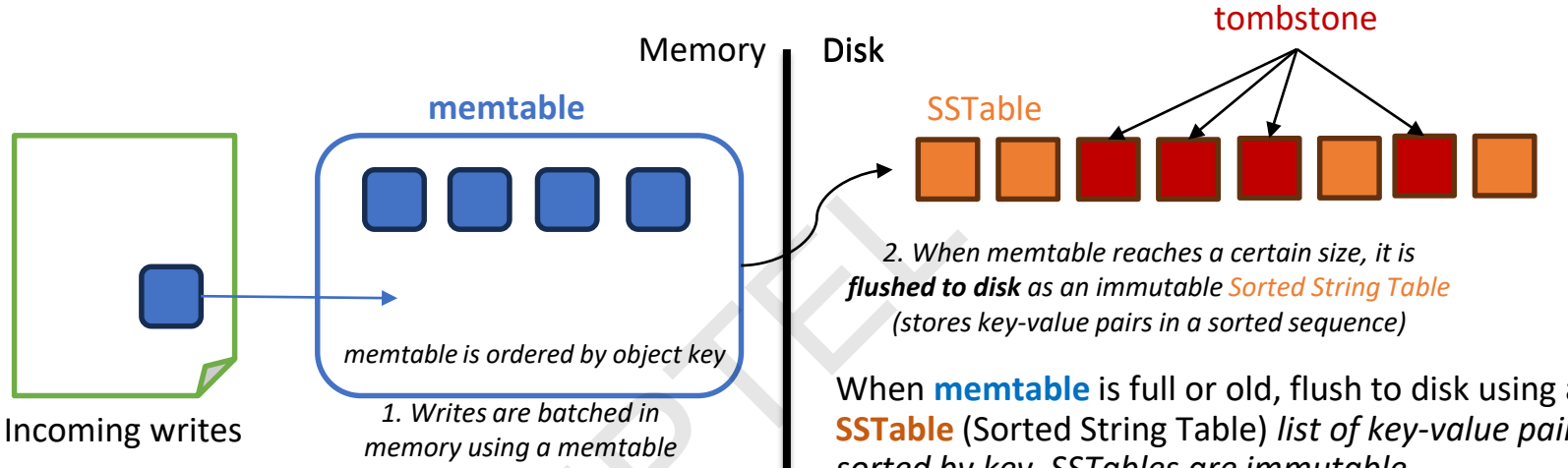
RackInferringSnitch
The **RackInferringSnitch** determines the proximity of nodes by rack and datacenter, which are assumed to correspond to the 3rd and 2nd octet of the node's IP address, respectively. This snitch is best used as an example for writing a custom snitch class (unless this happens to match your deployment conventions).

Transcript Notes

PropertyFileSnitch
This snitch determines proximity as determined by rack and datacenter. It uses the network details located in the **cassandra-topology.properties** config file. When using this snitch, you can define your datacenter names to be whatever you want. Every node in the cluster is described in the `cassandra-topology.properties` file, and this file should be exactly the same on every node in the cluster.

Ec2Snitch
used for simple cluster deployments on Amazon EC2 where all nodes in the cluster are within a single region.
In EC2 deployments , the region name is treated as the datacenter name and availability zones are treated as racks within a datacenter.
For example, if a node is in the **`us-east-1` region** then **`us-east`** is the datacenter name and **`1`** is the rack location. (Racks are important for distributing replicas, but not for datacenter naming.)
this snitch does not work across multiple regions because private of IPs
If we are using only a single datacenter, we do not need to specify any properties.
If we need multiple datacenters, we can set the **`dc_suffix`** options in the configuration file.

Cassandra Read/Write



On receiving a write

1. Log it in disk commit log (for failure recovery)
2. Make changes to appropriate *memtables*

memtable – (In-memory)

- *Typically append-only data structure (fast)*
- Cache that can be searched by key
- Write-back as opposed to write-through

When **memtable** is full or old, flush to disk using an **SSTable** (Sorted String Table) *list of key-value pairs, sorted by key. SSTables are immutable.*

- Index file: An SSTable of *(key, position in data sstable) pairs* And
- Bloom filter is used *(for efficient search)*

Transcript Notes

Writes in Cassandra:

- Writes Need to be **lock-free and fast** (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
  - Coordinator may be per-key, or per-client, or per-query
  - Per-key Coordinator ensures writes for the key are serialized
- Coordinator uses Partitioner to send query to all replica nodes for a given key
- When replicas respond, the coordinator returns an acknowledgement to the client

Incoming Writes on **Nodes, On receiving a write,**

1. Logs it in commit log (for failure recovery) on disk
2. Make changes to appropriate *memtables (Writes are batched in memory using a memtable)*
   - **memtable** – (is held In-memory)
- *Typically append-only data structure (fast)*
- Cache that can be searched by key
- It uses Write-back as opposed to write-through

Transcript Notes

- In a write-through strategy, every write operation that occurs memory/cache is immediately mirrored or written through to the main storage.
- In a write-back strategy, write operations are first performed in the memory/cache , and the corresponding data in the main storage is updated later. The actual write to the main storage is deferred until it becomes necessary to evict the data from the cache or during a flush operation. Write-back caching can provide lower latency for write operations compared to write-through, as writes are initially performed in the faster cache memory.

- *When memtable reaches a certain size, it is **flushed to disk** as an immutable Sorted String Table (stores key-value pairs in a sorted sequence) – the* Flushing operation is sequential and fast
- Each SSTable represents a small chronological set of incoming changes

- For deleting a key, the corresponding SSTables are marked as **tombstone**

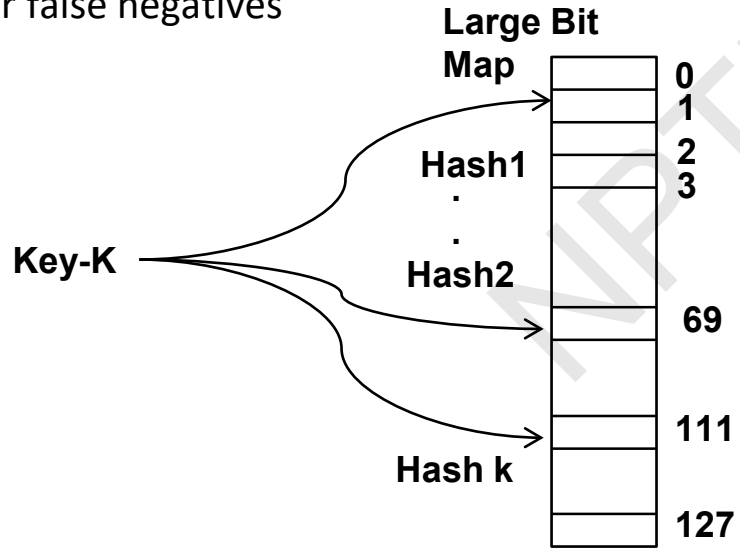
Transcript Notes

- For reading a key, the key is first looked up in the memtable and then in the SSTables starting at the most recently flushed SSTable
    - **Coordinator can contact X replicas (e.g., in same rack)**
        - Coordinator sends read to replicas that have responded quickest in past
        - When X replicas respond, coordinator returns the latest-timestamped value from among those X
        - X can be configured according to Replication Factor
    - **Coordinator also fetches value from other replicas**
        - Checks consistency in the background, initiating a **read repair** if any two values are different
        - This mechanism seeks to eventually bring all replicas up to date
    - Reads may be slower than writes as a row may be split across multiple replicas, hence multiple Stables needs to be touched (read from)

- However, as the more and more SSTables are flushed to disk, it takes increasingly longer time to look up a key (even though the SSTables are sorted).
- Also, the number of outdated keys (**tombstones**) keeps increasing overtime taking up unnecessary disk space.
- Hence, parodic **merging and compaction** process runs in the background that merges SSTables (by merging updates for a key) and discards tombstones.
- The merging process is simple and efficient as the SSTables are sorted.

Cassandra Bloom Filters

- Compact way of representing a set of items
- Checking for existence in set is cheap
- Some probability of false positives: an item not in set may check true as being in set
- Never false negatives



On insert, set all hashed bits.

On check-if-present, return true if all hashed bits set.

- **False positives**

False positive rate low

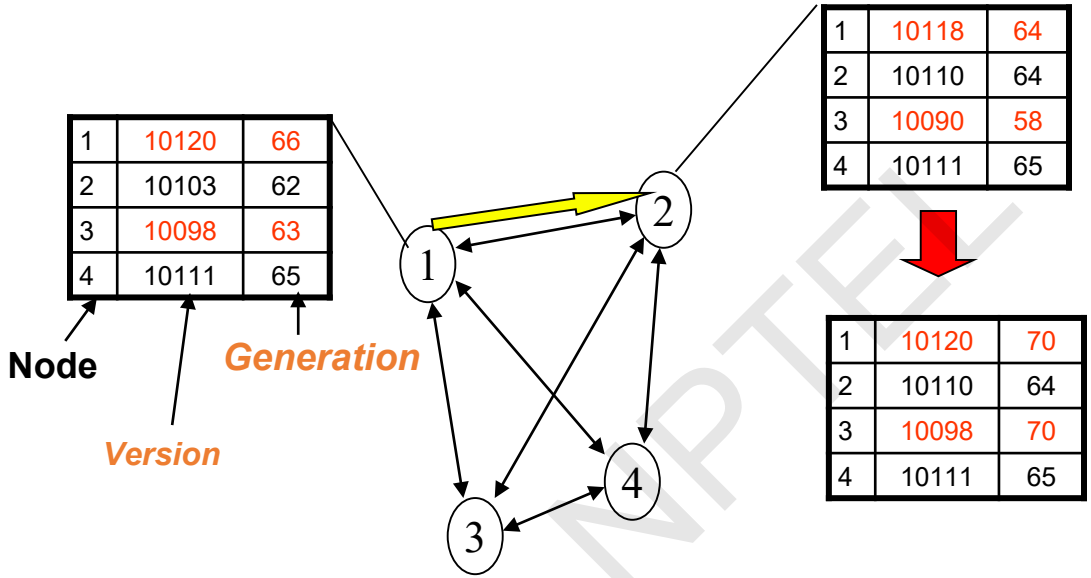
- **m=4 hash functions**
- **100 items**
- **3200 bits**
- **FP rate = 0.02%**

Transcript Notes

- Bloom filter is a Compact way of representing a set of items. A **Bloom filter** is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set.

- Checking for existence in set is cheap
- Some probability of false positives: an item not in set may check true as being in set -
- Never false negatives- False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set".
- Elements can be added to the set, but not removed - the more items added, the larger the *probability of false positives becomes*

Cluster Membership – Gossip-Style



E.g.,

- **Node 1 sends gossip message (with information about 4 nodes) to Node 2 – the current time at node 2 is 70**
- **Node 2 compares and updates its information (marked in red entries in the table)**
- **Node 2 Updates information about node 1 and node 3 since they have newer timestamp**

Transcript Notes

- Every member in a cluster maintains a list of member nodes – that needs to be updated as nodes join and leave
- The replication protocols and dataset partitioning rely on knowing which nodes are alive and dead in the cluster so that write and read operations can be optimally routed. In Cassandra, this kind of liveness information is shared in a distributed fashion through a failure detection mechanism based on a **gossip** protocol. Gossip is used to propagates basic cluster bootstrapping information such as endpoint **membership** and internode network protocol versions.
- In Cassandra's gossip system, nodes exchange state information not only about themselves but also about other nodes they know about.
- This information is versioned with a vector clock of (***generation***, ***version***) tuples where the ***generation*** is a monotonic timestamp and ***version*** is a logical clock that increments roughly every second. These logical clocks allow Cassandra gossip to ignore old versions of cluster state just by inspecting the logical clocks presented with gossip messages.

Transcript Notes

- Every node in the Cassandra cluster runs the gossip task independently and periodically. Every second, every node in the cluster:

    1. Updates the local node's heartbeat state (the **version**) and constructs the node's local view of the cluster gossip endpoint state.
    2. Picks a random other node in the cluster to exchange gossip endpoint state with.
    3. Probabilistically attempts to gossip with any unreachable nodes (if one exists)
    4. Gossips with a seed node if that didn't happen in step 2.

- Certain nodes are designated as seed nodes at the time of bootstrapping a cluster. Any node can be a seed node, and the only difference between seed and non-seed nodes is that seed nodes are allowed to bootstrap into the ring without seeing any other seed nodes. Furthermore, once a cluster is bootstrapped, seed nodes become hotspots for gossip due to step 4 above.

Cassandra Cluster Membership

- Every node in the Cassandra cluster runs the gossip task independently and periodically. Every second, every node in the cluster:

    1. Updates the local node's heartbeat state (the version) and constructs the node's local view of the cluster gossip endpoint state.
    2. Picks a random other node in the cluster to exchange gossip endpoint state with.
    3. Probabilistically attempts to gossip with any unreachable nodes (if one exists)
    4. Gossips with a seed node if that didn't happen in step 2.

- Certain nodes are designated as seed nodes at the time of bootstrapping a cluster. Any node can be a seed node, and the only difference between seed and non-seed nodes is that seed nodes are allowed to bootstrap into the ring without seeing any other seed nodes. Furthermore, once a cluster is bootstrapped, seed nodes become hotspots for gossip due to step 4 above.

Cassandra Failure Detection

- Gossip forms the basis of ring membership, but the **failure detector** ultimately makes decisions about if nodes are UP or DOWN.

- Every node in Cassandra runs a variant of the *Phi Accrual Failure Detector*, in which every node is constantly making an independent decision about - if their peer nodes are available or not. This decision is primarily based on received heartbeat state.

- ϕ represents the Inter-arrival times for gossip messages which decides the threshold time for if a node has failed
  - $\phi(t) = -log_{(10)}P_{Later}(t)$

- t indicates the time elapsed since last gossip message received from an endpoint
- $P_{Later}(t)$ indicates probability of receiving a heartbeat after t units of time which determines the likelihood of endpoint failure

Cassandra Failure Detection

- Latest version of Cassandra implements a modified version of this test assuming exponential distribution of gossip messages. ϕ is approximated as,
  $\phi(t) = 0.43429645 \times \frac{t}{m}$
  Where m is the arithmetic mean the most recent gossip message inter-arrival times

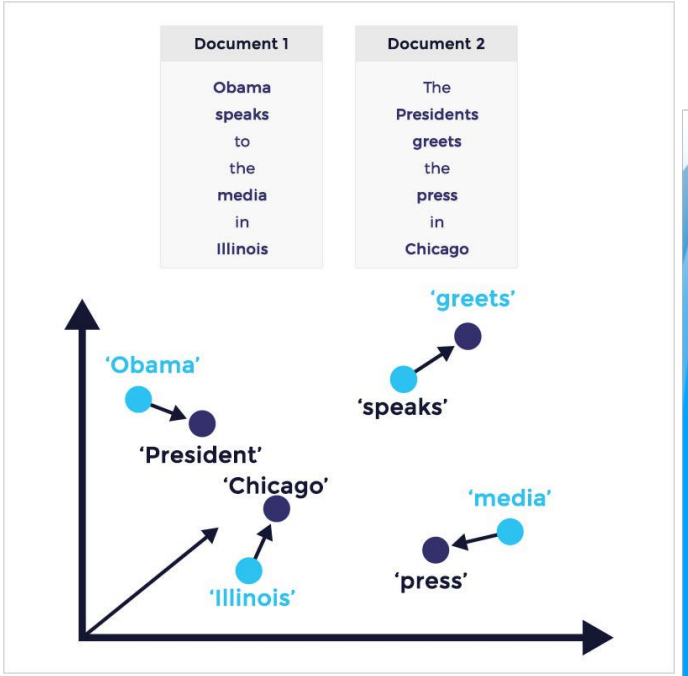
- For example, if a node does not see an increasing heartbeat from a node for a certain amount of time, the failure detector "convicts" that node, at which point Cassandra will stop routing reads to it (writes will typically be written to hints).

- When the node starts “heartbeat” again, Cassandra will try to reach out and connect, and if it can open communication channels it will mark that node as available.

- UP and DOWN state are local node decisions and are not propagated with gossip. Heartbeat state is propagated with gossip, but nodes will not consider each other as UP until they can successfully message each other over an actual network channel.

Cassandra Vector Search

- Vector Search is a *new feature* added to Cassandra 5.0. It is a powerful technique for finding relevant content within large document collections and is particularly useful for AI applications.
- **Vector Search**: For performing similarity comparisons in a machine learning model, we need the ability to store embeddings vectors.
- Embeddings are compact representations of text or images as high-dimensional vectors of floating-point numbers. For text processing, embeddings are generated by feeding the text to a machine learning model such as a Large Language Model.
- A new **vector data type** is added to CQL to support Vector Search. It is designed to save and retrieve embeddings vectors.



Cassandra V/S RDBMS

- MySQL is one of the most popular (and has been for a while)
- On > 50 GB data
- **MySQL**
  - Writes 300 ms avg
  - Reads 350 ms avg
- **Cassandra**
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster
- What's the catch? What did we lose?

CAP Theorem

CAP Theorem

- **Proposed by Eric Brewer (Berkeley)**
- Subsequently proved by Gilbert and Lynch (NUS and MIT)
- In a distributed system you can satisfy atmost 2 out of the 3 guarantees:

1. **Consistency:** all nodes see same data at any time, or reads return latest written value by any client
2. **Availability:** the system allows operations all the time, and operations return quickly
3. **Partition-tolerance:** the system continues to work in spite of network partitions

Availability

- **Availability =** Reads/writes complete reliably and quickly.
- Why is Availability Important?
  - User cognitive drift
  - SLAs (Service Level Agreements)

Consistency

- **Consistency =** all nodes see same data at any time, or reads return latest written value by any client.

- Why is Consistency Important?

  - Multiple points of access – banking applications

  - Multiple Concurrent accesses – ticket booking applications

Partition-Tolerance

- **Partitions** can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working

- Partitions can also occur within a datacenter, e.g., a rack switch outage

- Still desire system to continue functioning normally under this scenario

Transcript Notes

- Availability implies that Reads/writes complete reliably and quickly.

- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue. At Amazon, each added millisecond of latency implies a $6M yearly loss.
- Real-time Analytics is important for many business organizations
- User cognitive drift: If more than a second elapses between clicking and material appearing, the user's mind is already somewhere else

- SLAs (Service Level Agreements) written by service providers predominantly deal with latencies faced by clients. Failure to meet SLA requirement may cause monetary losses to service providers
- Consistency = all nodes see same data at any time, or reads return latest written value by any client.

- Multiple points of access – When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.

- Multiple Concurrent accesses – When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.
- Partitions can happen across datacenters when the Internet gets disconnected
- Internet router outages
- Under-sea cables cut
- DNS not working
- Partitions can also occur within a datacenter, e.g., a rack switch outage
- We Still desire system to continue functioning normally under any of outage scenarios

CAP Theorem Fallout

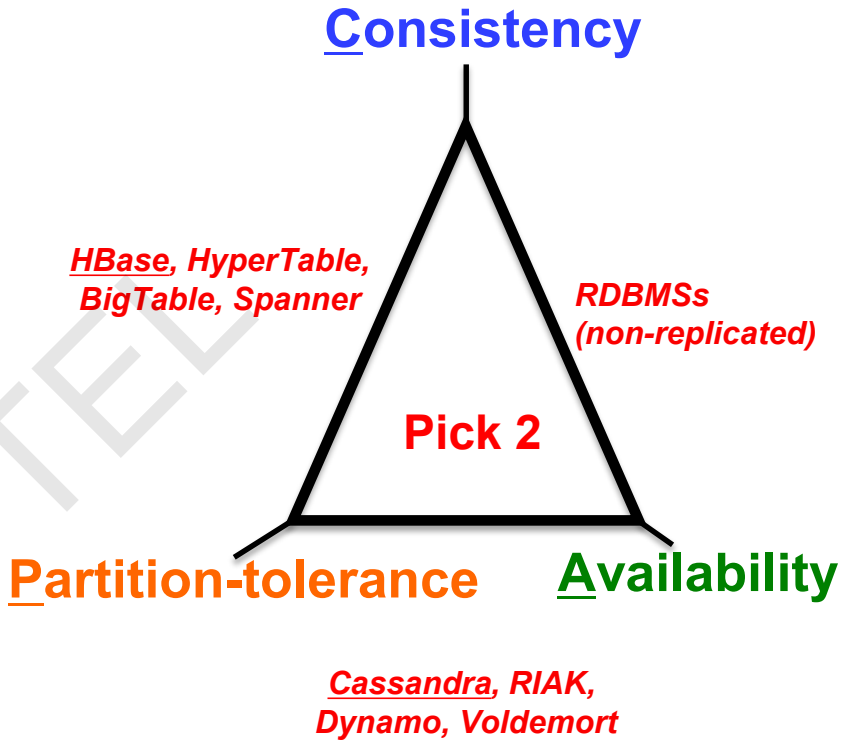
- Since **partition-tolerance** is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability

- **Cassandra**
  - Eventual (weak) consistency, Availability, Partition-tolerance

- **Traditional RDBMSs**
  - Strong consistency over availability under a partition

CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve **at most two of C, A, and P**.
- When partition-tolerance is important, you have to choose between consistency and availability



Eventual Consistency

- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
    - **Moving "wave" of updated values lagging behind the latest values sent by clients, but always trying to catch up.**
- May still return stale values to clients (e.g., if many back-to-back writes).
- But works well when there a few periods of low writes – system converges quickly.

Transcript Notes

- Eventual consistency means If all writes (to a key) stop, then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
- There will be a Moving “wave” of updated values lagging behind the latest values sent by clients, but always trying to catch up.
- Consistency achieved eventually but no immediately
- if many back-to-back writes occurs – it May still return stale values to clients
- But works well when there a few periods of low writes – system converges quickly.

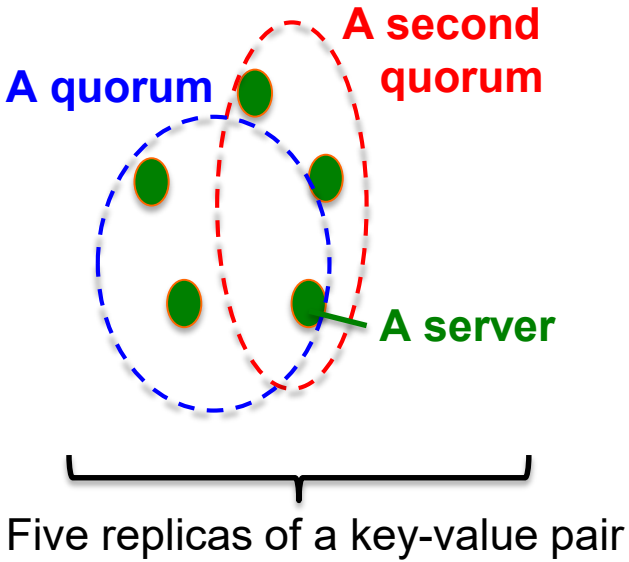
Relational Database vs. Key-value stores

- While RDBMS provide **ACID**
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

- Key-value stores like Cassandra provide **BASE**
  - **B**asically **A**vailable **S**oft-state **E**ventual Consistency
  - Prefers Availability over Consistency

Consistency Levels

- Cassandra has **consistency levels**
- Client is allowed to choose a consistency level for each operation (read/write)
  - **ANY:** any server (may not be replica)
  - **ALL:** all replicas
  - **ONE:** at least one replica
  - **QUORUM:** quorum across all replicas in all datacenters (Quorum = majority)
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency



Transcript Notes

- To ensure consistency in a distributed environment, Cassandra allows users to specify a consistency level for read and write operations.
- For example:
- If the replication factor is 5 (meaning each piece of data is stored on 5 nodes), a quorum would require at least 3 out of 5 nodes to respond
- The replica set is a group of nodes that store copies of the same data for fault tolerance and high availability.
- When performing a read or write operation in Cassandra, you can specify a consistency level, which determines how many nodes must acknowledge the operation for it to be considered successful. Common consistency levels include:

- ANY: any nodes (may not be replica) may respond – its is the Fastest: coordinator caches write and replies quickly to client
- ALL: All nodes must respond. Ensures strong consistency, but slowest
- ONE: at least one node needs to respond. Faster than ALL, but cannot tolerate a failure
- QUORUM: A majority of nodes in the replica set must respond (at least half + 1). In the context of Cassandra, a quorum represents a majority of nodes within a replica set. The number of nodes required to form a quorum depends on the total number of replicas and the specified consistency level.

- Any two quorums intersect
- Client 1 does a write in red quorum
- Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums are faster than ALL, but still ensure strong consistency

- The choice of consistency level affects the trade-off between consistency and availability in a distributed system. A higher consistency level ensures stronger consistency but may increase latency, especially in the presence of network partitions or node failures. In contrast, a lower consistency level can provide lower latency but may sacrifice some level of consistency.

Quorums in Detail

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.
- **Reads**
  - Client specifies value of **R** (≤ N = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Quorums in Detail (Contd..)

- Writes come in two flavors
  - Client specifies W (≤ N)
  - W = write consistency level.
  - Client writes new value to W replicas and returns.
    Two flavors:
    - Coordinator blocks until quorum is reached.
    - Asynchronous: Just write and return.

Quorums in Detail (Contd.)

- R = read replica count, W = write replica count
- Two necessary conditions:
  1. **W+R > N**
  2. **W > N/2**
- Select values based on application
  - **(W=1, R=1):** very few writes and reads
  - **(W=N, R=1):** great for read-heavy workloads
  - **(W=N/2+1, R=N/2+1):** great for write-heavy workloads
  - **(W=1, R=N):** great for write-heavy workloads with mostly one client writing per key

Cassandra Consistency Levels

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - **QUORUM:** quorum across all replicas in all datacenters (DCs)
    - Global consistency, but still fast
  - **LOCAL_QUORUM:** quorum in coordinator's DC
    - Faster: only waits for quorum in first DC client contacts
  - **EACH_QUORUM:** quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies

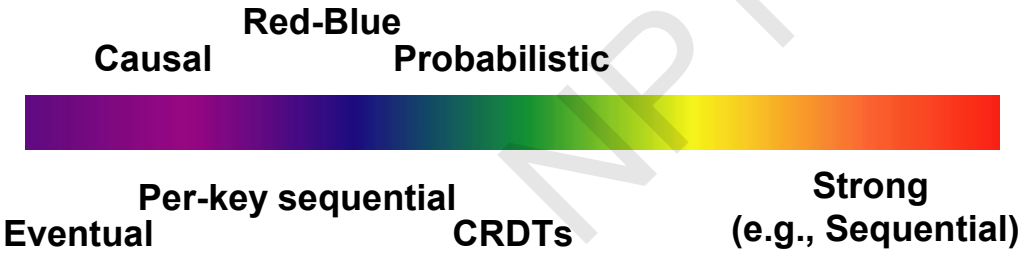
Consistency Solutions

Consistency Solutions

- Cassandra offers **Eventual Consistency**
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

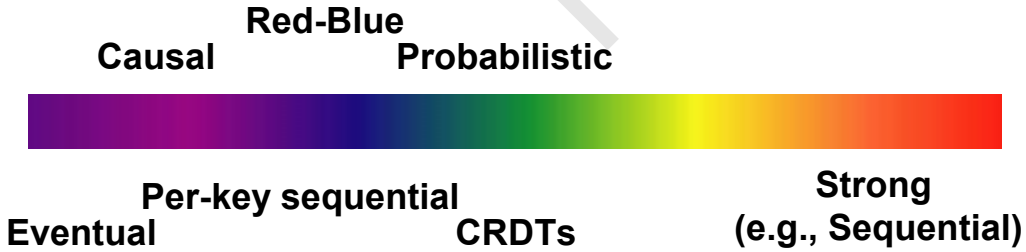
Newer Consistency Models

- Striving towards strong consistency
- While still trying to maintain high availability and partition-tolerance



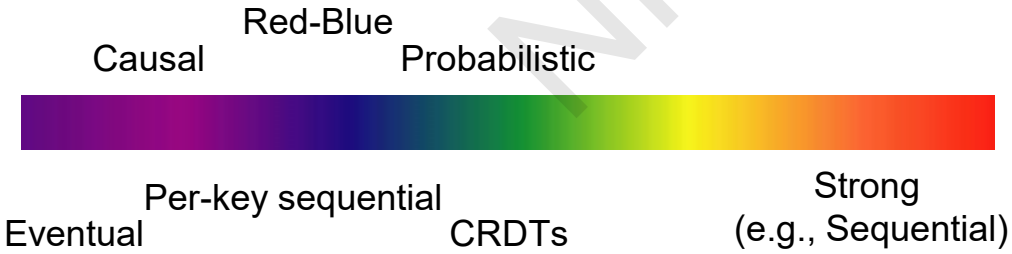
Newer Consistency Models (Contd.)

- **Per-key sequential**: Per key, all operations have a global order
- **CRDTs** (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]
  - E.g., value == int, and only op allowed is +1
  - Effectively, servers don't need to worry about consistency



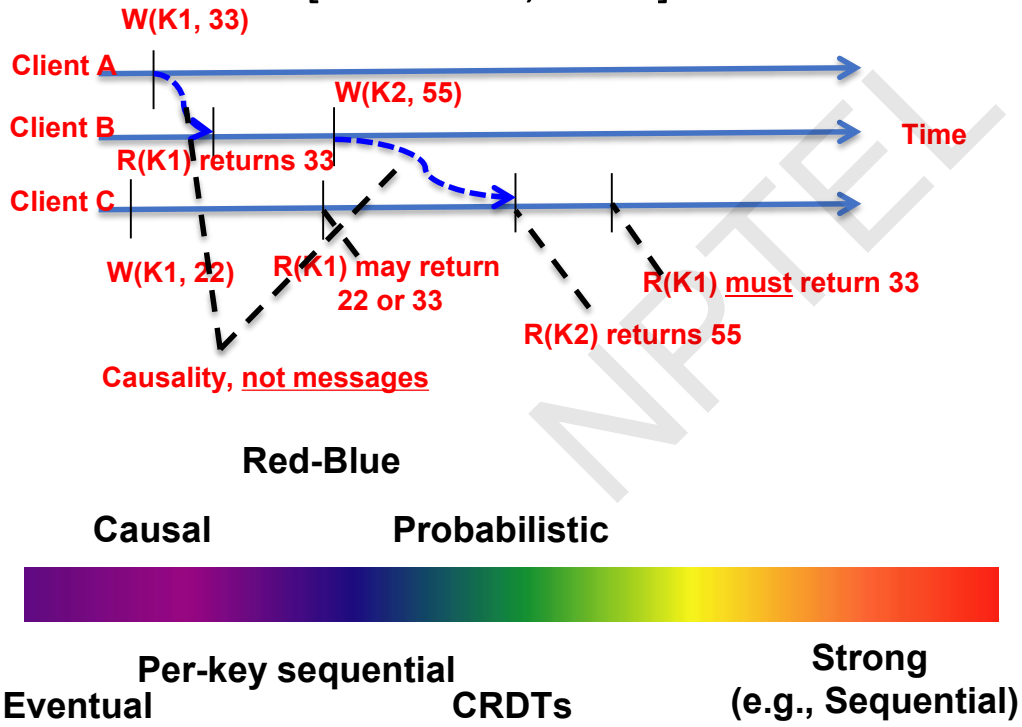
Newer Consistency Models (Contd.)

- **Red-blue Consistency**: Rewrite client transactions to separate operations into red operations vs. blue operations [MPI-SWS Germany]
  - Blue operations can be executed (commutated) in any order across DCs
  - Red operations need to be executed in the same order at each DC



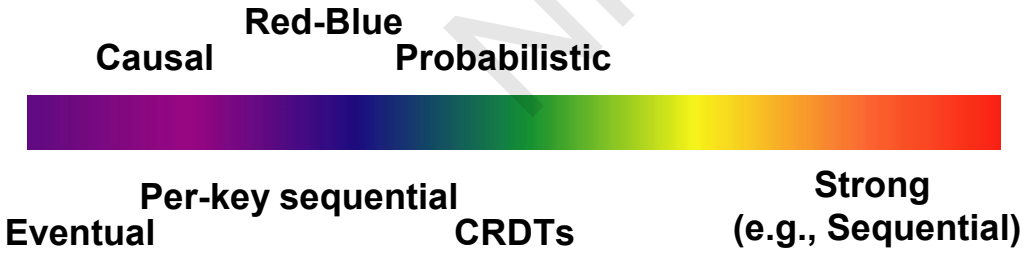
Newer Consistency Models (Contd.)

Causal Consistency: Reads must respect partial order based on information flow [Princeton, CMU]



Which Consistency Model should you use?

- Use the lowest consistency (to the left) consistency model that is “correct” for your application
  - Gets you fastest availability



Strong Consistency Models

- **Linearizability**: Each operation by a client is visible (or available) **instantaneously** to all other clients
  - Instantaneously in real time
- **Sequential Consistency** [Lamport]:
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
  - After the fact, find a "reasonable" ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- Transaction ACID properties, example: newer key-value/NoSQL stores (sometimes called **"NewSQL")**
  - Hyperdex [Cornell]
  - Spanner [Google]
  - Transaction chains [Microsoft Research]

Conclusion

- We have discussed the NoSQL database management approach and compared it to Traditional Databases (RDBMSs)
- We have also discussed the design of Cassandra and different consistency solutions.
- CAP theorem
- **BASE: B**asically **A**vailable **S**oft-state **E**ventual Consistency
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency

Thank You!

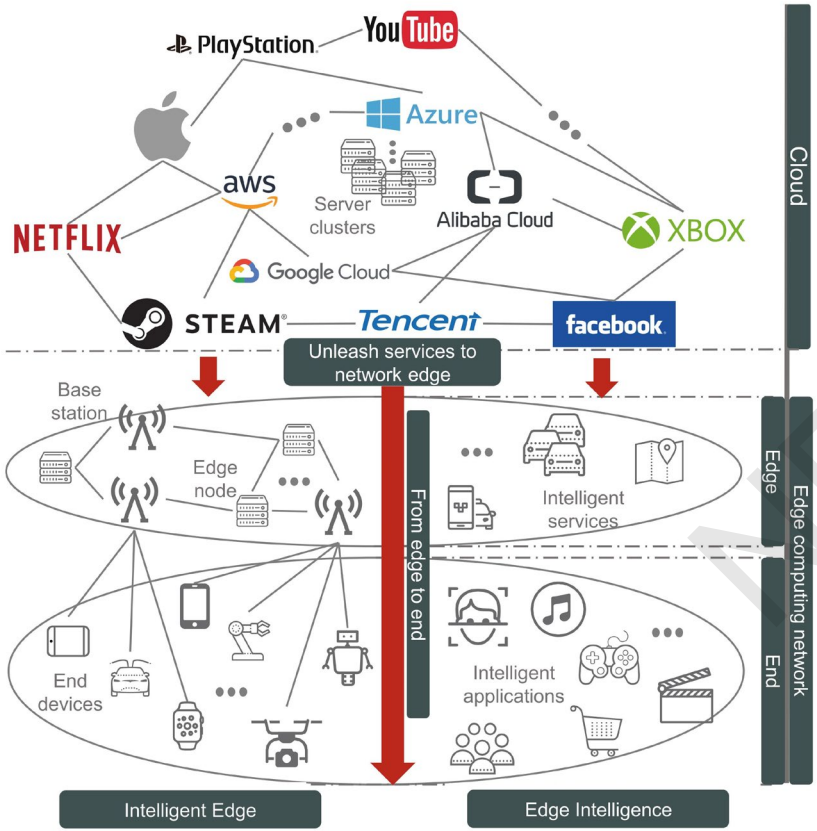
Week 3 Lecture 3

0 mins

Edge AI - Intelligence at the Edge

Dr. Rajiv Misra, Professor
Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Edge AI – Intelligent-Edge v/s Edge-Intelligence



- ✓ Ultra low latency
- ✓ Data Reduction
- ✓ Enhanced privacy
- ✓ Reliability
- ✓ Scalability

Figure Source: *https://viso.ai/edge-ai/edge-intelligence-deep-learning-with-edge-computing/*

Transcript Notes

Edge computing is gradually being combined with Artificial Intelligence (AI), benefiting each other in terms of the realization of intelligence at the edge i.e., Edge AI.

Edge AI can be classified into two categories, i.e., **edge-intelligence (EI)** and **intelligent-edge(IE)** as depicted in the **Fig**

To be specific,

edge intelligence brings the AI services from the cloud to the edge as much as possible, thus enabling various distributed, low-latency, and reliable intelligent services. Considering that AI is functionally necessary for the quick analysis of huge volumes of data and extracting insights, there exists a strong demand to integrate edge computing and AI, which gives rise to edge intelligence.

edge intelligence studies how to run AI models on edge. It is a framework for training and inference of AI models within the device, edge and cloud continuum.

It aims at extracting insights from massive and distributed edge data with the satisfaction of algorithm performance, cost, privacy, reliability, efficiency, etc.

Therefore, it can be interpreted as **AI on the edge.**

intelligent edge aims to incorporate AI into the edge for dynamic, adaptive edge maintenance and management.

intelligent edge focuses on providing a better solution to constrained optimization problems in edge computing with the help of effective AI technologies.

Here, AI is used to endow edge with more intelligence and optimality. Therefore, it can be understood as **AI for the Edge or intelligence-enabled edge computing**.

Transcript Notes

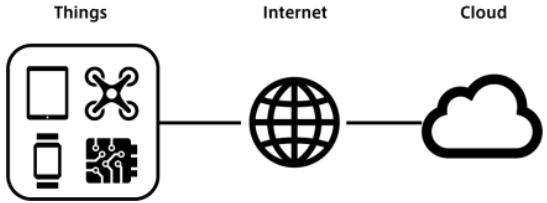
Traditionally, AI services (such as training and Inferencing) were performed on the cloud. However, it is not suitable for tasks that require real-time analysis and quick decision making (such as autonomous vehicles). Furthermore, moving high-volume of data acquired by a large number of sensors and IoT devices becomes inefficient and time consuming. In such scenarios, edge computing can play crucial role in enabling intelligence services close to the users or the end-devices.

Notable of the benefits of edge-computing in this regard can be noted as follows:

1. **Ultra low latency and response time** - significantly lower than cloud, It enables **Real-time data analysis and decision making**
2. **Data Reduction -** Reduces the volume of data to be transferred on the network (up-stream) and uses the bandwidth efficiently, therefore, optimizes the cost as well. Data is locally generated and can be cached on the edge; this also increases the network throughput.
3. **Enhanced privacy and security** – Data is processed locally, preserving privacy and reducing security threats.
4. **Reliability or Offline functionality** i.e., can provide services without the help of cloud – devices can function even when the connection to the cloud/ internet is down
5. **Scalability –** Edge is scalable by design; it scales well with the amount of data to be processed at the edge.

Edge-Intelligence for IoTs

Recapitulate IoTs



An **Internet of Things (IoT)** is a network that connects uniquely identifiable “Things” to the Internet. In other words, IoT is a network of internet connected devices embedded in everyday objects enabling sending & receiving data and provide control.

The Things have sensing/actuation; and may have programmability capabilities as well.

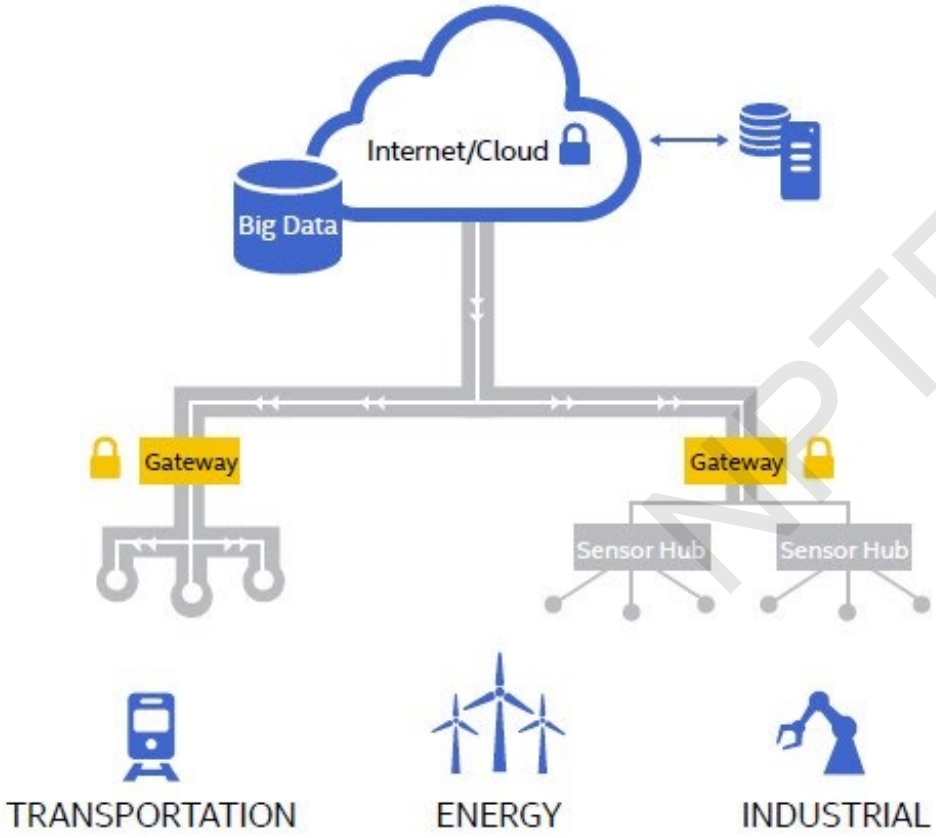
The Main task of IoT networks are:

1. **Gather information from things**
   - monitoring: state information
   - control: command enforcement
2. **Send information back and forth** remote locations (typically cloud)
3. **Store and aggregate** information
4. **Analyze** information to improve system knowledge
5. **Take decisions**, in a human-assisted or autonomous manner

Transcript Notes

- An **Internet of Things (IoT)** is a network that connects uniquely identifiable "Things" to the Internet. In other words, IoT is a network of internet connected devices embedded in everyday objects enabling sending & receiving data and provide control. **The Things** have sensing/actuation; and may have programmability capabilities as well. The sensing/actuating covers everything from legacy industrial devices to robotic camera systems, water-level detectors, air quality sensors, accelerometers, and heart rate monitors.
- Through identification and sensing, information about the Thing can be collected (telemetry); and the state of the Thing can be changed from anywhere and at anytime (actuation/control).
- Gathered data is analyzed to acquire a more **in-depth knowledge of the things**, typically composed of **geographically distributed** and **heterogeneous** things.
- **The Main task of IoT networks are:**
- **Gather information from things** and send commands to things
  - monitoring: state information
  - control: command enforcement
- **Send information back and forth** remote locations (edge, cloud)
- **Store and aggregate** information
- **Analyze** information to improve system knowledge
- **Take decisions**, in a human-assisted or autonomous manner

Typical Cloud-based IoT Architecture



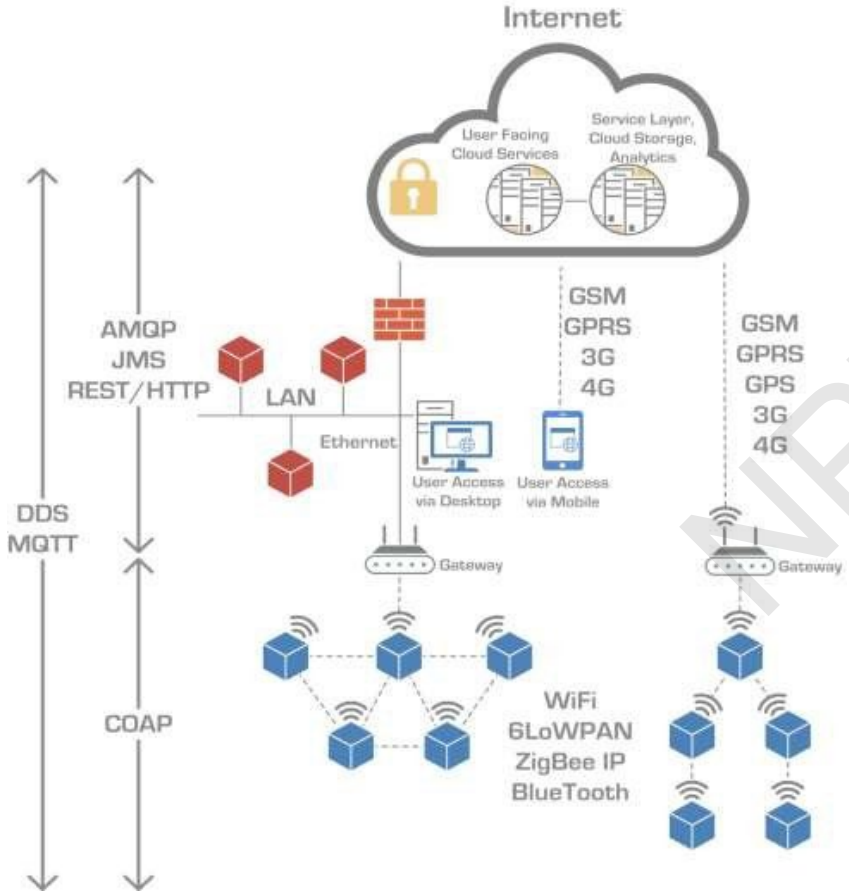
- Several heterogeneous **things**, e.g., sensors and actuators
- Multiple **gateways** geographically close to sensors/actuators
  - directly interact with things
  - dispatch data to/from the Internet
- Server-side remote **applications** stored in the Cloud and managing data

Transcript Notes

In a traditional Cloud-based IoT architecture, we have the following:

1. several heterogeneous **things**, e.g., sensors and actuators, may be geographically disturbed across various regions
2. Multiple **gateways** located geographically close to the **things -** gateways directly interact with the things and dispatch data to/from the Internet (cloud)
3. **Cloud**: Server-side remote **applications** are stored in the Cloud and all the data management, analytics or business-logic related tasks are performed on the cloud.

Typical Cloud-based IoT Architecture



- Gateways provide **protocol translation**
- Gateways can also provide data **buffering, aggregation, filtering.**
- Gateways also provide **security**, scalability, service discovery, **geo-localization**, **billing**, etc.

Transcript Notes

What is a gateway?

In an IoT network, a **gateway** plays a crucial role in facilitating communication between the end-devices and the central cloud. The primary purpose of an IoT gateway is to manage and optimize data flow between the devices and the cloud.

An IoT gateway facilitates **protocol translation** by acting as an intermediary between IoT devices that use different communication protocols, depending on their manufacturers and specific functionalities. Gateways can translate and bridge communication between devices using different protocols. This allows for integration of diverse devices within an IoT ecosystem.

- Gateways can also provide pre-processing, **security**, scalability, service discovery, geo-localization, billing services, etc.

Transcript Notes

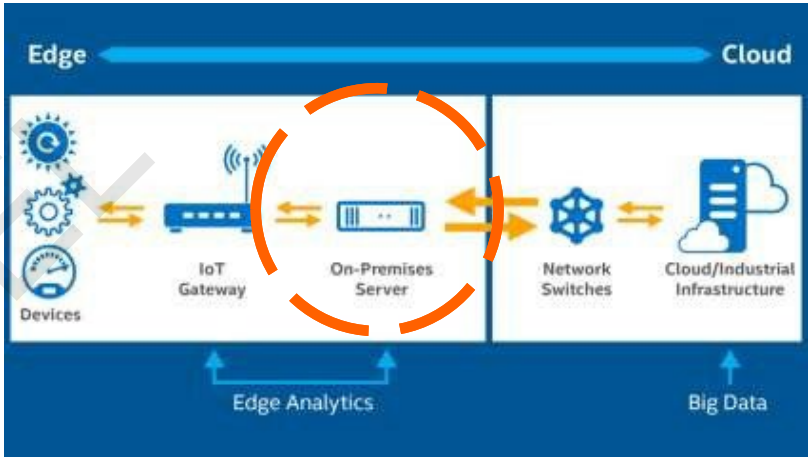
- for e.g in gateway Pre-processing we can have :
  - data **buffering**: e.g., temporarily store data to wait for connectivity or to increase efficiency
  - data **efficiency**: e.g., read temperature every 1s, but only per-minute average is sent
  - data **aggregation**: e.g., read water level from different silos, but only the sum is sent
  - data **filtering**: e.g., send temperature values only if grater than 25°C
- If there is no gateway, things have to send/receive data on their own. The **constrained devices** will have reduced set of capabilities, e.g.,
  - no security since cryptography would be CPU-intensive
  - no data buffering, filtering or aggregation
  - no programmability

For example, a pump might contain a many sensors and actuators that feed data into a data aggregation device that also digitizes the data. This device might be physically attached to the pump. An adjacent gateway device or server would then process the data. Intelligent gateways can build on additional, basic gateway functionality by adding such capabilities as analytics, malware protection, and data management services.

From Cloud Computing to Edge Computing

- First evolution wave: **IoT Cloud Computing** architecture
  - most of the computation on the Cloud
  - **only gateways are deployed close to things**
  - gateways perform **few and simple tasks**

- Second evolution wave: IoT **Edge** Computing architecture
  - **relatively powerful gateway**
  - **on-premise edge servers close to things**, but between gateways and the Cloud
  - **complex analytical tasks** on-premise, before sending data to the Cloud



Transcript Notes

- First evolution wave: **IoT Cloud Computing** architecture
  - most of the computation happens on the Cloud
  - **only gateways are deployed close to things**
  - gateways perform **few and simple tasks**
- However, Cloud models are not designed for the volume, variety, and velocity of data that the IoT generates
- Second evolution wave: IoT **Edge** Computing architecture
  - **gateway is relatively powerful (and possibly intelligent)**
  - **on-premise edge servers are located close to things**, but between gateways and the Cloud
  - **complex analytical tasks are now perfomed** on-premise, before sending data to the Cloud

For example, In a predictive maintenance task for an Industrial pump, rather than passing on the raw vibration data for the pumps, you could aggregate and convert the data, analyze it, and send only projections as to when each device will fail or need service.

Here's another example: You might use machine learning at the edge to scan for anomalies that identify impending maintenance problems that require immediate attention. Then you could use visualization technology to present that information using easy-to-understand dashboards, maps, or graphs. Highly integrated compute systems, such as hyper-converged infrastructure, are ideally suited to these tasks because they're relatively fast, and easy to deploy and manage remotely.

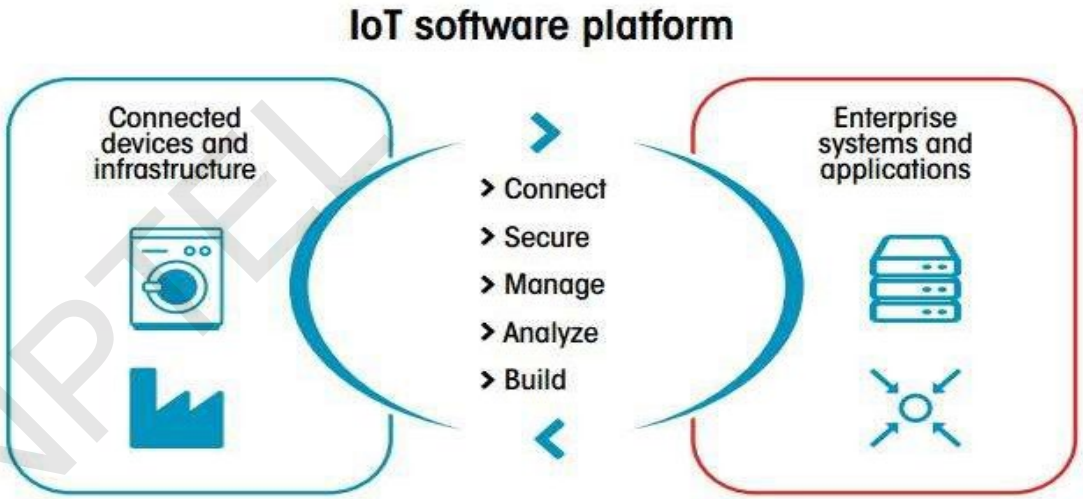
Transcript Notes

Therefore, most of the AI/ML enabled applications are now deployed at the edge instead of cloud.
The edge layer delivers three essential capabilities in this regard:

1. **Local data processing:** huge volume of data is collected at the **extreme edge**: vehicles, ships, factory floors, roadways, railways, etc.
   In order to deal with increasing amount of data generated by sensors, most of the business logic is now deployed at the edge layer instead of cloud to ensure low-latency and faster response time.
2. **Filtered data transfer to cloud**: thousands or millions of things across a large geographic area are generating data
   Only a subset of the data generated by sensors is sent to the cloud after aggregating and filtering the data at the edge. This Edge Computing approach significantly saves the bandwidth and cloud storage.
3. **Faster decision-making:** when it is necessary to analyze and **act on data promptly**, in milliseconds.
   Since most of the decision-making is now taking advantage of artificial intelligence, the edge layer is becoming the perfect destination for deploying machine-learning models trained in the cloud.

Recapitulate IoT Platforms

- IoT Platform between things/devices and business applications
  - **connect** devices to gather information
  - **secure** communication with devices and applications
  - **manage** devices to control their behavhior
  - **analyze** data, e.g., with AI
  - **build** applications, also interacting with CRM/ERP/…



Transcript Notes

- Internet of Things (IoT) platforms play a crucial role in the deployment, management, and analysis of IoT devices and data. These platforms serve as the middleware that connects the physical world of devices to the digital world of software applications, enabling seamless communication, data processing, and control.

Transcript Notes

Device Management:

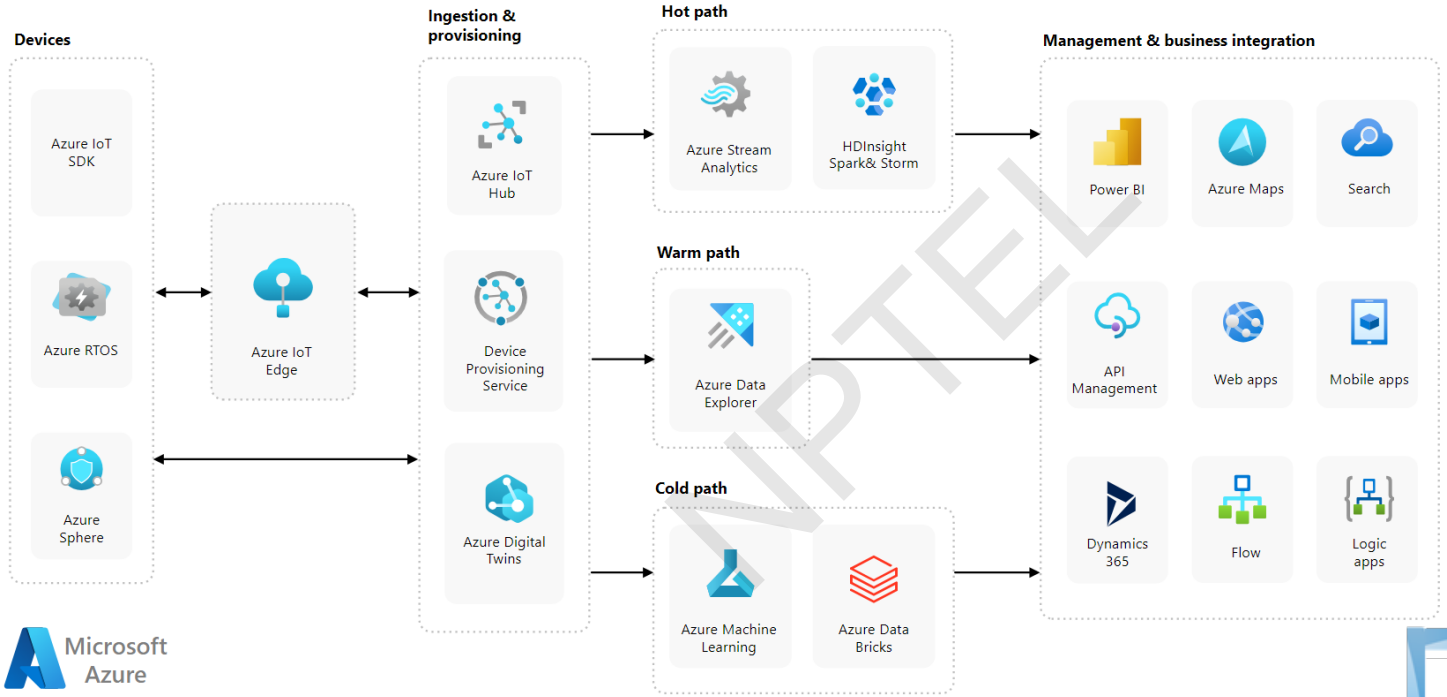
•*Functionality:* IoT platforms provide tools for registering, provisioning, and managing devices. This includes capabilities for firmware updates, remote monitoring, and diagnostics.

- **to connect** devices to gather information from. enable security mechanisms like TLS/SSL for secure device registration
- **to secure** communication with devices and applications - management involves protocols like MQTT, CoAP, or HTTP for communication. These Communication protocols are lightweight and efficient to accommodate the constraints of IoT devices.
- **to manage** devices to control their behavior. Integration with existing enterprise systems, databases, and other third-party services is a crucial aspect of IoT platforms. APIs (RESTful, MQTT, etc.) facilitate integration, and standards like OData or OpenAPI may be employed. Message brokers and middleware solutions are also common for system integration. It enables Managing the entire lifecycle of devices, from onboarding to decommissioning, is essential for long-term IoT deployments.
- **to analyze** data, e.g., with AI/ML. It include tools for real-time data processing, analytics, and visualization. This involves filtering, aggregating, and analyzing data streams from multiple devices.
- **to build** applications, also interacting with CRM and ERP platforms
  - Customer relationship management (**CRM**) is a technology for managing all your company's relationships and interactions with customers and potential customers.
  - Enterprise resource planning (**ERP)** – which is a type of business management software.
- **to scale :** IoT platforms must be able to scale horizontally to accommodate an increasing number of devices and data volume. Containerization (e.g., Docker), orchestration (e.g., Kubernetes), and microservices architecture contribute to the scalability of IoT platforms.

Several IoT Platforms

- Amazon Web Services (AWS) IoT
- Microsoft Azure IoT
- Mindsphere by Siemens
- EdgeXFoundry
- Google Cloud Platform
- ThingWorx IoT Platform
- … and many more

Azure IoT Platform: Overview



Microsoft Azure

Source: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/iot

Transcript Notes

Azure IoT is a platform provided by Microsoft, and it encompasses both PaaS (Platform as a Service) and SaaS (Software as a Service) components, depending on the specific services within the Azure IoT suite.

The major components of Azure IoT platform are:

- **Devices:** Represents the components required to be run on IoT devices
- Azure IoT supports a large range of devices such as,
  - microcontrollers running Azure RTOS and Azure Sphere
    - **Azure RTOS (Real Time Operating System)** is an embedded development suite including a small but powerful operating system for resource-constrained devices. Azure RTOS supports the most popular 32-bit microcontrollers and embedded development tools.
    - Azure Sphere is a development suit that includes the **Azure Sphere OS** along with some other tools.
  - developer boards like MX Chip and Raspberry Pi.
  - Azure IoT also supports **smart server gateways** capable of running custom code using the IoT SDKs.

Transcript Notes

- **Ingestion & Provisioning**
  - Azure IoT Hub is a cloud gateway service that can securely connect and manage devices. Azure IoT Hub is a core PaaS offering within Azure IoT. It is a scalable and fully managed service that facilitates bidirectional communication between IoT applications and the devices it manages. IoT Hub acts as a message broker, handling device-to-cloud and cloud-to-device communication, as well as device management and authentication.
  - Azure IoT Hub Device Provisioning Service (DPS) enables automatic just-in-time provisioning that helps to register a large number of devices in a secure and scalable manner. It provides secure communications and complete control through a specific authentication for each device.
  - Hub allows sending both the **telemetry data from devices to cloud** and **commands to change devices' behavior from the cloud** . The hub has Built-in Routing functionality with rules to automatically dispatch messages. It also stores the c**onfiguration of devices using metadata and state information** which can be queried from the up-stream.
  - IoT Hub supports the MQTT, AMQP and HTTPS protocols.

  - **Azure IoT Central (SaaS):** Azure IoT Central is a SaaS solution that simplifies the creation, deployment, and management of IoT applications. It provides pre-built templates and tools for quickly creating IoT solutions without the need for extensive coding or infrastructure management. Azure IoT Central is more focused on providing a ready-to-use SaaS experience for **IoT applications.**

Transcript Notes

- Azure Digital Twins is a PaaS that enables virtual models of real world systems. A digital twin is a virtual model of a real-world environment that is driven with data from business systems and IoT devices. for e.g buildings, infrastructure, or entire cities. These digital twins provide a virtual counterpart to the physical entities, allowing for real-time monitoring, analysis, and simulation of their behavior.

- When devices are connected to the cloud, there are several services that assist with ingesting data. At a high level, there are three ways to process data: **hot path, warm path, and cold path**. The paths differ in their requirements for latency and data access.

  - The **hot path** analyzes data in near-real-time as it arrives. Hot path telemetry must be processed with very low latency.
  - The hot path typically uses a stream processing engine. The output of stream processing might trigger an alert or be written to a structured format that can be queried using analytical tools. services such as Azure Stream Analytics are often used for stream processing.

  - We can also use tools like apache-kafka and Databricks for stream processing functionality.

Transcript Notes

- The **warm path** analyzes data that can accommodate longer delays for more detailed processing. Tools that are more in line with warm path such as data lake, factory data, factory synapse, databricks use ter Azure Functions. e.g., Azure Data Explorer is used for storing and analyzing large volumes of data.

- The **cold path** performs batch processing at longer intervals, like hourly or daily. The cold path typically operates over large volumes of data, which can be stored in Azure Data Lake Storage which is commonly used for storing and analyzing massive amounts of data, including structured, semi-structured, and unstructured data.

- Results from the cold path don't need to be as timely as in the hot or warm paths. Cold path is more batch-oriented, hot path will process the message as it hits the system while cold path really processes the messages as they accumulate on the system and rather being triggered by the message itself what it allows for is data to be accumulated over a period of time and then typically on a trigger that is timer based it will then take whatever data has been accumulated and process that data in batch. We can use Azure Machine Learning or Azure Databricks to analyze cold data.

Transcript Notes

- Azure Machine Learning (Azure ML) is a PaaS offering within the Microsoft Azure cloud platform. Azure ML provides a set of cloud-based services and tools designed to simplify the process of building, training, and deploying machine learning models.

- Azure Databricks is a PaaS offering. It is a cloud-based Apache Spark and analytics platform provided by Microsoft Azure in collaboration with Databricks. Azure Databricks simplifies the process of setting up, configuring, and managing Apache Spark clusters for big data processing and analytics.

- The takeaway from this is that hot path is real-time warm path is going to be more often smaller workloads that are going to be rating on smaller time scales like 5 minutes, 10 minutes, 15 minutes or an hour and cold path is going to be larger workloads that are going to be operating over long periods of time. It might be five minutes if there's a lot of data it could be an hour it could be a day could be a week.

Transcript Notes

- **Management and Business Integration -** Business integration actions might include:
  - Storing informational messages.
  - Raising alarms.
  - Sending email or SMS messages.
  - Integrating with business applications such as customer relationship management (CRM) and enterprise resource planning (ERP).

- many services are provided by Azure for management and business integration such as
  - Power BI for modeling & visualizing the data and using artificial intelligence to make data-driven decisions.
  - Azure Maps for creating location-aware web and mobile applications by using geospatial APIs, SDKs, and services like search, maps, routing, tracking, and traffic.
  - Azure Cognitive Search provides a search service over varied types of content. Cognitive Search includes indexing, AI enrichment, and querying capabilities.
  - Azure App Service helps to deploys web applications in a scalable way
  - etc

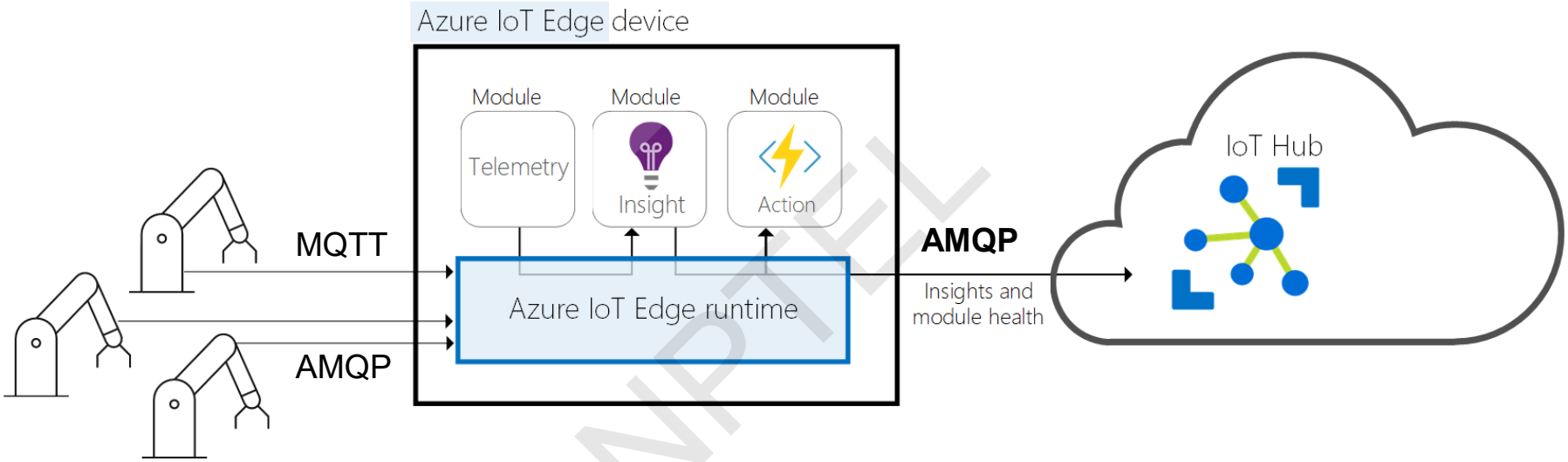
Transcript Notes

Finally ----------

- **IoT Edge:** Azure IoT edge is a service to deploy and manage applications over remote devices in their locations. Azure IoT Edge is a hybrid solution that combines both PaaS and on-premises capabilities. It extends Azure IoT services and analytics to edge devices, allowing processing and analysis of data closer to the source to drive better business insights and enable offline decision making. Azure IoT Edge enables the deployment of containerized workloads to edge devices, enhancing edge computing capabilities. Azure IoT Edge is a part of Azure IoT Hub which is a managed service hosted in the cloud that acts as a central message hub for communication between an IoT application and its attached devices.

- For e.g., On the edge of a network we have a local area network and a bunch of devices that are emitting elementary and events that are ultimately sent back to the cloud. However, in some cases you might want to put some kind of preprocessor in place that will do some filtering and aggregation and some other enhancements on the data closer to where the devices are. For example, we can run anomaly detection workloads at the edge to respond as quickly as possible to emergencies happening on a production line. If we want to reduce bandwidth costs and avoid transferring terabytes of raw data, you can clean and aggregate the data locally then only send the insights to the cloud for analysis.

Azure IoT Edge



- Three main components:
  - **Edge Modules**
  - **Edge Runtime**
  - **Cloud interface**

Source: *https://learn.microsoft.com/en-in/training/modules/introduction-iot-edge/3-how-iot-edge-works*

Transcript Notes

- Azure IoT Edge comprises **three components:**

- **1. IoT Edge modules** are units of execution implemented as Docker compatible containers. IoT Edge modules can run business logic at the edge and are capable of running other azure services, third-party services, or any custom code. These services or code can be deployed as **Modules** on the Edge devices and execute locally on those devices. You can develop custom IoT Edge modules in several languages, with SDKs for Python, Node.js, C#, Java, and C. The Azure IoT Edge Marketplace also offers some prebuilt modules.
- The modules can be configured to communicate with each other to create a pipeline for data processing. Modules can run offline if needed. Every module is made of 4 conceptual elements:

- An Image: package containing **the software of the module**
- An Instance: **unit of computation** that runs the image on the device; it is started by IoT Runtime.
- An Identity: information about **credentials and permissions** associated with each module.
- A Twin: **JSON document** that stores metadata regarding the **status of a module and configuration**.

Transcript Notes

- **2. IoT Edge runtime** runs on each IoT Edge device and manages the runtime and communication for the modules deployed to each device. The IoT Edge runtime is a collection of programs that turn a device into an IoT Edge device. The IoT Edge runtime is responsible for the following functions on IoT Edge devices:
  - Install and update workloads on the device.
  - Maintain security standards on the device.
  - Ensure that IoT Edge modules are always running.
  - Report module health to the cloud for remote monitoring.
  - Manage communication between:
    - Downstream devices and IoT Edge devices
    - Modules on an IoT Edge device
    - An IoT Edge device and the cloud
    - IoT Edge devices

Transcript Notes

- The responsibilities of the IoT Edge runtime fall into two categories: **communication** and **module management**. These two roles are performed by two components
    - 1. The ***IoT Edge agent*** which deploys and monitors the modules i.e., it pulls down the container orchestration manifest from the cloud, so IoT Edge knows which modules to run. It's responsible for instantiating modules, ensuring that they continue to run, and reporting the status of the modules back to IoT Hub.
    - The *IoT Edge **hub*** module manages inter-module communication and communication between the device and Azure IoT Hub in the cloud. Messages route from one module to the next with JSON configuration. IoT Edge encrypts and streams real-time industrial data to IoT Hub by using AMQP or MQTT protocols. It acts as a local proxy for IoT Hub by exposing the same protocol endpoints as IoT Hub. This consistency means that clients can connect to the IoT Edge runtime just as they would to IoT Hub. The IoT Edge hub isn't a full version of IoT Hub running locally. IoT Edge hub silently delegates some tasks to IoT Hub. For example, IoT Edge hub automatically downloads authorization information from IoT Hub on its first connection to enable a device to connect. After the first connection is established, authorization information is cached locally by IoT Edge hub. Future connections from that device are authorized without having to download authorization information from the cloud again.
- Both the **agent** and the **edge-hub** are modules, just like any other module running on an IoT Edge device. They're sometimes referred to as the *runtime modules*.

Transcript Notes

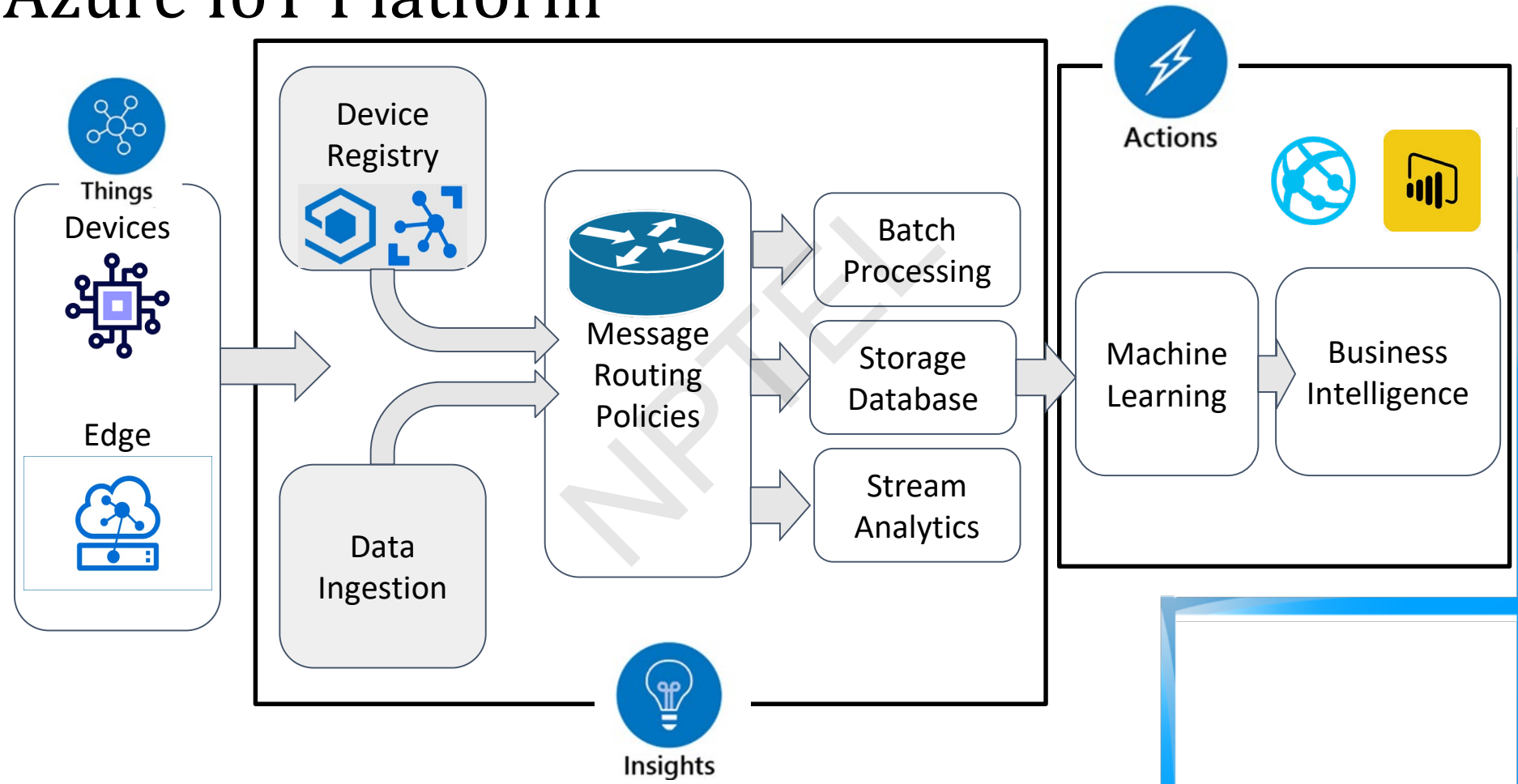
- The runtime once installed turns a device into an Edge device and gives them many features:
- Ensures that modules are running and report module health to the cloud for monitoring
- Manages communications between edge devices and normal devices, upstream with the cloud, and downstream with machines

- **3. IoT Edge cloud interface** enables you to monitor and manage overall lifecycle at scale for a diverse set of devices, which could be geographically scattered.
- It's challenging to manage the software lifecycle for millions of IoT devices that are often different makes and models or in diverse locations. Workloads are created and configured for a particular type of device, deployed to all of the devices, and monitored to catch any misbehaving devices. These activities can't be done on a per device basis and must be done at scale.
- Consider the case of deploying complex event processing or machine learning on edge devices. If you want to implement machine learning on edge devices, you must first train the model in the cloud. After training the model, you need to deploy the trained model to a diverse range of edge devices often across geography. Once deployed, these models will run often offline on the device. You would also need to update the model periodically. By encapsulating the models in docker compatible containers, IoT Edge can manage the end-to-end cycle of deployment for machine learning on IoT. IoT Edge runtime and the cloud interface can monitor the status of the machine learning modules.In the absence of the IoT Edge, the developer would need to create the added functionality of maintaining the module.

Transcript Notes

- Again, To reduce the bandwidth usage, the IoT Edge hub optimizes how many actual connections are made to the cloud. IoT Edge hub takes logical connections from modules or downstream devices and combines them for a single physical connection to the cloud. Clients think they have their own connection to the cloud even though they're all being sent over the same connection. The IoT Edge hub can either use the AMQP or the MQTT protocol to communicate upstream with the cloud, independently from protocols used by downstream devices.
- IoT Edge hub can determine whether it's connected to cloud IoT Hub . If the connection is lost, IoT Edge hub saves messages and updates locally. Once a connection is reestablished, it syncs all the data.

Azure IoT Platform



Transcript Notes

- Azure IoT is internally organized in **3 levels of abstraction / layers**
- **Things**: data is generated from **devices (sensors and actuators)**, has a brief processing through **edge applications** and is sent trough some **connectivity service** to the next layer.
  - The generated data is going to be acquired and ingested into the cloud.
  - The devices are further connected to the edge and the edge acts as a gateway abstracting the devices that are at the lowest level of the spectrum and that actually connects to the cloud.
- **Insights:** In this stage the data is processed, and stored in different kinds of storage or databases, aggregated and analyzed through **stream analytics** and **machine learning tools** to derive some insights.

Transcript Notes

- Now on the cloud side we have two touch points for the edge or the devices.
- One is the **device registry** that is primarily used for onboarding the devices and it is the repository of devices.
- Every device that is connected to the IOT platform has an identity within the device registry. The authentication authorization and the metadata of the devices is stored in the device registry.
- Consider an enterprise corporate directory scenario where the device registry like an LDAP of devices, you can query to get a lot of metadata and useful information about every device connected to the platform.
- The public cloud pass also called IOT pass exposes a data ingestion endpoint. This is the high velocity, high throughput endpoint where the sensor data gets streamed. This typically could be Kafka or event hubs or Amazon kinases or Google cloud publisher-subscriber interface, etc
- So, it is the pipe that basically acquires the data and passes on to the data processing pipeline

Transcript Notes

- Now both the device registry and data ingestion endpoints are connected to a **message routing policy.**
- A message routing policy which will define how this data is going to be split between real-time processing and batch processing and how the raw data is stored and how the processed data is going to be stored.
- This is the place where you actually create a rules engine or you basically create some kind of policy that is going to define how the data flows.
- For example, some data needs to be batch process, where you first collect and then process, in some cases you need to perform real-time stream analytics.

- The batch processing layer which is also called as cold path analytics and stream analytics layer which is also called as hot path analytics.

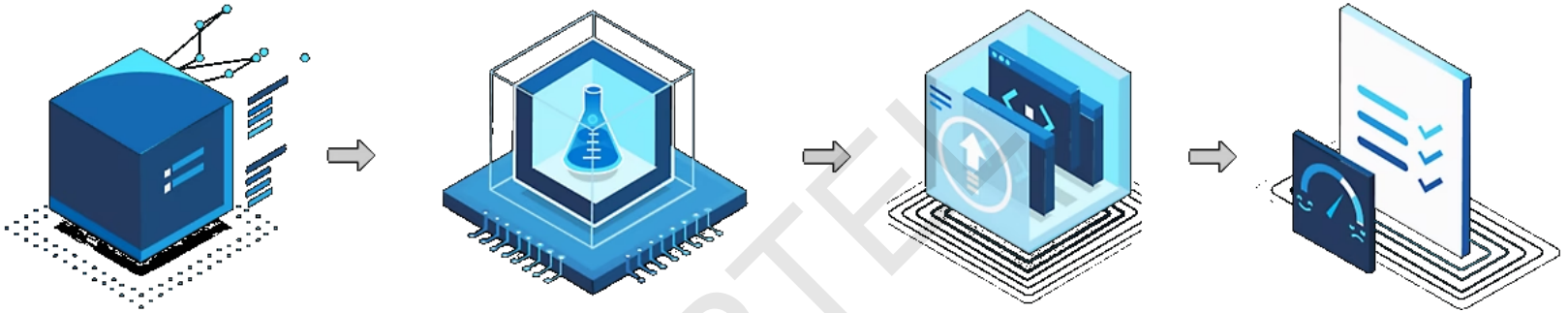
- In other words, when you are performing queries on data as it comes that is called the hot path analytics and if you are storing and processing the data over a period of time it is called the cold path analytics.
- Now both the raw data which is going or about to go to either batch processing or stream analytics is first persisted in a time series database or an unstructured database and even the output from the batch processing and stream analytics gets persisted in the same database.
- Then we have storage and databases for persisting the raw sensor data and/or the processed data.

Transcript Notes

- **Actions**: In this layer the **insights** are visualized into graphs and **dashboards** that can help companies to take some actions e.g., operational decisions, business decisions etc.
  - i.e., data from the data storage is fetched and we can apply machine learning algorithms to train models which can then perform tasks such as anomaly detection and predictive analytics from the data that is coming in.
  - Finally, all of that is fed into an enterprise Business Intelligence Platform, that runs dashboards & alerts. The entire visualization happens on the business intelligence layer.

Machine Learning at the Edge

- A typical ML workflow has four stages

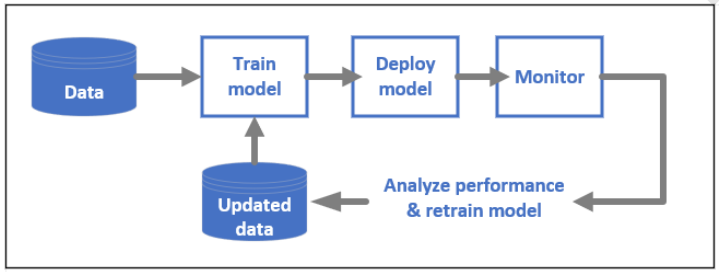


Data Preparation
acquiring and preparing data for training

Model Preparation
building and training ML models

Model Deployment
deploying models on the edge runtime

Monitoring
monitoring and maintaining ML models



ML model lifecycle

Transcript Notes

SOURCE: https://azure.microsoft.com/en-in/products/machine-learning

The General Edge ML model development and deployment process follows a cyclical, iterative pattern. A typical ML workflow has 4 stages as shown in the figure

1. Data preparation

- **Data acquisition – In case of IoT,** collect data from sensors and devices like Camera, Lidar, industrial sensors etc
- **Data cleaning and labeling -** Clean the acquired data i.e., filtering out corrupted data and missing values. Labeling the data if required, using annotation tools like VoTT(Visual Object Tagging Tool**)**,
- **Data Exploration** - Use analytics engines for data exploration (exploratory analysis); this helps to further organize the data according to different ML task requirements.
- **Datasets -** prepare the data in form of datasets and store them in an appropriate storage for future stages

2. Model Preparation – Building and training AI/ML models

- Building the ML models using frameworks like pytorch, tensorflow, scikit-learn, CNTK etc
- Training and validating the models on stored datasets – training is usually performed in the cloud where computation resources are available in terms of CPUs/GPUs/TPUs, cloud can provide ample resources to train large ML models such as a deep neural network.
- trained models are stored in a model-repositories (or a model registry) on the cloud (to be used later for inferencing)

Transcript Notes

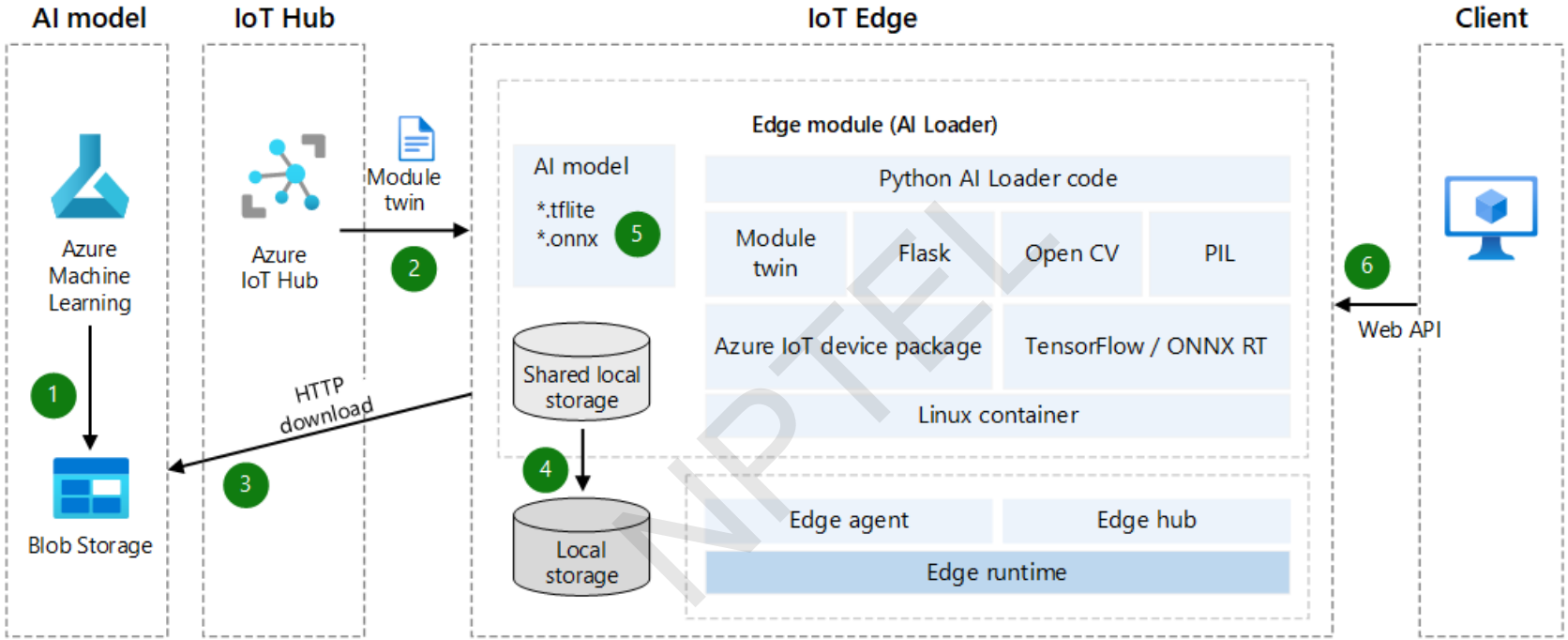
3. Model Deployment – Inferencing at the Edge Deploying the trained models on the edge runtime
To bridge the gap between the cloud and edge, innovations in chip designs offers purpose-built accelerator that speed up model inferencing at the edge significantly. Chip manufacturers such as Qualcomm, NVIDIA and ARM have launched specialized chips that speed up the execution of ML-enabled applications.
These modern processors contain GPUs on the edge devices which can improve the performance of deep learning models by accelerating the inference process.

- to deploy an ML model, first it needs to be converted to work with a particular **IoT Edge runtime.**
- **the edge-runtime** can be a Docker-based system that can deploy the ML models as containers (eg-Azure IoT Edge runtime)
- Model conversion usually includes optimizations like faster inference and smaller model footprint.
- The process differs for various frameworks **such as TFLite, Snapdragon Neural Processing Engine, NVIDIA DeepStream etc, which makes it possible to run hardware-accelerated inference at the edge.**

4. Monitoring and Maintenance – includes tasks such as,

- **Monitoring and analysis –** Tracking model usage, analyse resources utilization and model related data.
- **Error analysis -** Detect drift in inferencing and maintain model accuracy. **-** Debug models and optimize AI model accuracy.
- **Update and Improve –** Improve existing models using newer data, perform fine-tuning using real-time data,
- **Auditing -** Tracing machine learning artifacts for compliance management and ethical use
- **Cost control -** management of resource consumption; e.g., automatic shutdown after resource quota is consumed

Inferencing at the Edge



machine learning inference on an IoT Edge device

Source: *https://learn.microsoft.com/en-us/azure/architecture/guide/iot/machine-learning-inference-iot-edge*

Transcript Notes

- AI on the edge is one of the most popular edge scenarios. Implementations of this scenario include image classification, object detection, body, face, and gesture analysis, and image manipulation. This architecture describes how to use IoT Edge to support these scenarios.
- We can improve AI accuracy by updating the AI model, but in some scenarios the edge device network environment isn't good. For example, in the wind power and oil industries, equipment might be located in the desert or the ocean. In such a case, IoT Edge **module twins** are used to implement the dynamically loaded AI model.
- **module twins** are JSON documents that store module state information including metadata, configurations, and conditions. Azure IoT Hub maintains a module twin for each module connected to IoT Hub via the edge-hub.
- Module twins store module-related information that:
    - Modules on the device and cloud IoT Hub can use to synchronize module conditions and configuration.
    - can be used to query and target long-running operations.
- The concept of "twin" comes from the idea that each IoT device or module has a digital twin in the cloud, which represents the current state, metadata, and desired state of the device or module. Hence a digital twin is a cloud-based representation of a physical device or module. Module twins include properties that reflect the current state of the associated module. These properties can be updated by the module itself or by external applications. The module twin is associated with the identity of a specific module within an IoT device. Module twins include desired properties, reflecting the desired configuration for the associated module. External applications or cloud services can update these properties to signal the desired state. Cloud-to-Device communication allows external applications to send messages or updates to a specific device or module and In this way the Desired properties of a module can be updated through C2D communication, triggering the module to adjust its configuration to match the desired state.

Transcript Notes

- We have seen earlier that IoT Edge modules are based on Docker Containers. An image for an IoT Edge module in an AI environment typically has a size of at least 1 GB, so incrementally updating the AI model is important in a narrow-bandwidth network. That consideration is the main focus of this slide.
- The idea is to create an IoT Edge AI module that can load ML and DL frameworks like TensorFlow Lite or Open Neural Network Exchange (ONNX) based AI models.
- The edge modules can be enabled with a web API so that they can be used by other applications or modules.
- The solution includes 3 steps:
    - 1. Enable AI inference on edge devices.
    - 2. Minimize the network cost of deploying and updating AI models on the edge. This can save money for you or your customers, especially in a narrow-bandwidth network environment.
    - 3. Create and manage an AI model repository in an IoT edge device's local storage. This will help to achieve almost zero downtime when the edge device switches AI models.

Transcript Notes

- **Dataflow (marked in green circled numbers)**

1. The AI model is uploaded to Azure Blob Storage or a web service. The model can be a pre-trained TensorFlow Lite or ONNX model or a model created in Azure Machine Learning. The IoT Edge module can access this model and download it to the edge device later.
2. Azure IoT Hub syncs device module twins automatically with AI model information. The sync occurs even if IoT Edge has been offline. (In some cases, IoT devices are connected to networks at scheduled hourly, daily, or weekly times to save power or reduce network traffic.)
3. The loader module monitors the updates of the module twins via API. When it detects an update, it gets the machine learning model SAS token and then downloads the AI model. SAS stands for shared access signature which is mechanism that allows access to containers.
4. The loader module saves the AI model in the shared local storage of the IoT Edge module
5. The loader module loads the AI model from local storage via the TensorFlow Lite or ONNX API.
6. The loader module starts a web API that receives the input data (e.g, an image as binary data) via POST request and returns the results in a JSON file.

- To update the AI model, you can upload the new version to Blob Storage and sync the device module twins again for an incremental update. There's no need to update the whole IoT Edge module image. In this solution, an IoT Edge module is used to download an AI model and then enable machine learning inference.

Edge ML Platform Example Scenario

Workload description

IoT Edge Runtime

Azure IoT Hub

Insight Module
Azure Cognitive Service

Custom Code
Data pre-processing & transformation

Action Module
Custom Code

Container Registry

Transcript Notes

Example scenario for How Edge ML platform (Azure IoT) provides services in terms of SaaS. Once edge-device is connected to the Hub, the modules are responsible for capturing and processing data.
Data can be captured using Kafka(High Volume) or simple telemetry (low volume data)

Step 1: developing and registering modules (which are containers images)
Assuming that Some modules are already available or have been already created by developers for processing data which are:

1. **Data preprocessing and Transformation**: This is a custom module created to pre-process and transform the incoming data from a security camera
2. **Insight Module**: - running the Azure cognitive services in the cloud – Azure Cognitive Services is a set of cloud-based services provided by Microsoft Azure that enables developers to add various artificial intelligence (AI) capabilities to their applications without requiring expertise in machine learning or AI. These services are designed to make it easy for developers to integrate features like computer vision, natural language processing, speech recognition, and more into their applications.
3. **Action Module**: This is also a custom module that decides which actions to take after data has been processed by insight module.

The container images are stored in some container registry. These containers images are ready to be deployed on edge or the cloud

Transcript Notes

Step 2: Create The Workload description or deployment manifest
Contains the information about All the workloads (containerized modules required to run the workload at the edge)
The ML platform (SaaS) will fetch the appropriate containers from the registry when the deployment manifest is provided to the edge runtime

Step 3: Target an IoT edge Runtime on an edge device
The runtime is actually responsible for deploying or running the required containers on the edge device. It fetches the appropriate containers as described in the workload description

Step 4: The Edge Runtime now performs ML (or analytics tasks) as defined in the modules. Furthermore, the monitoring data is sent to cloud via the IoT Hub if required for maintenance and improvement.

Conclusion

- In this lecture we discussed about enabling edge intelligence for the IoT.
- We discussed about Azure IoT Edge platform that provides various services that enable AI workloads such as Machine Learning and Deep Learning models to be deployed at the Edge devices using containerized runtimes.

Thank You!

References

- Harmon, Robert & Castro-Leon, Enrique & Bhide, Sandhiprakash. (2015). Smart cities and the Internet of Things. 485-494. 10.1109/PICMET.2015.7273174.