Cloud Computing (UE18CS352)
Unit 2

Aronya Baksy

February 2021

1 Introduction

- **Virtualization** is a framework for dividing a single hardware resource (compute or storage) into multiple independent environments.
- This is done by applying concepts such as h/w and s/w partitioning, emulation, etc.
- A **virtual machine** (VM) is a complete compute environment with its own processing capability, as well as memory and communication channels. It is an efficient, isolated duplicate of the physical machine, with the ability to run a complete operating system.
- A **hypervisor** (also called a **virtual machine monitor** or VMM) is a software layer that is responsible for creation and management of Virtual machines.

1.1 Why Virtual Machines

- **Operating System Diversity**: can run multiple different OS on a single machine
- **Rapid provisioning, server consolidation**: Allows for on-demand provisioning of hardware resources
- **High availability, Load Balancing**: the ability to live-migrate VMs ensures that these 2 aspects of cloud computing are handled
- **Encapsulation** of a single application's execution environment.

1.2 Qualities of a Hypervisor

- **Equivalence**: virtual and real machines should have a similar interface
- **Safety and isolation**: VMs should be isolated from one other, and also from the underlying physical hardware
- **Low performance overheads**: Virtual Machine should have similar performance to the physical machine.

2 Types of Virtualization

2.1 Type 1 virtualization

- Type 1 hypervisors are installed directly on top of a bare metal hardware, and they have direct control over hardware resources.
- Type 1 hypervisors behave like OSes with only virtualization functionality, and a limited GUI for administrators to configure system properties.
- Type 1 hypervisors offer simpler setup (provided that compliant hardware exists), more scalability, more security and higher performance than type 2 hypervisors.
- e.g.: Xen, Oracle VM (based on Xen), VMWare ESXi server, Microsoft Hyper-V

2.2 Type 2 virtualization

- This type of hypervisor runs within a host OS that runs on top of physical hardware. For this reason type 2 virtualization is also called **hosted** virtualization.
- They have interfaces to act as management consoles for all the deployed VMs
- Type 2 hypervisors offer simpler setup than type 1, but less scalability, larger performance overheads and less security than type 1.
- e.g.: VMWare Workstation, Oracle VirtualBox

2.3 Full Virtualization and Para-Virtualization

2.3.1 Para Virtualization

- In para-virtualization, the guest OS is modified in a way such that all privileged instructions in the kernel that are addressed to the hardware, are now replaced by **hyper calls** to the hypervisor.
- The Guest OS accesses privileged functions of the hardware through these hyper calls.
- Para-virtualization improves system performance and reduces overheads of virtualization, as all privileged instruction calls (which are handled using hyper calls) are all handled at *compile time*, instead of at run time.
- Disadvantage: the need for modifying the guest OS, and the resulting lack of portability across different

2.3.2 Full Virtualization

- The VMM simulates the hardware at a level that allows any *unmodified* guest OS to run in isolation on top of the host OS. It is also called *transparent virtualization*.
- Full virtualization is achieved using a combination of *binary translation* and *direct execution*.

3 Trap and Emulate Virtualization

- Instructions that cannot affect the state of the system (which is typically stored in control registers on the CPU labelled as CR0 to CR4) can be run directly by the hypervisor on the hardware.
- Sensitive instructions that change system state cannot be executed in user mode (ring 3). Such an attempt raises a **trap**, also called a *general protection fault*.
- The hypervisor emulates the *effect* of such sensitive instructions so that the guest OS still gets the *impression* that it is running in kernel mode when it is actually not.
- In trap-and-emulate virtualization, the:
  - Guest applications run on ring 3
  - Guest OS runs on ring 1
  - VMM runs on ring 0
- When a guest app in ring 3 issues a system call, an interrupt is issued to the guest OS in ring 1.
- The interrupt handler in the guest OS runs the system call routine. When a privileged instruction is encountered as part of this routine, the guest OS kernel issues an interrupt to the VMM.
- The VMM *emulates* the functionality of that privileged instruction, returns control to the guest OS.
- Essentially, trap-and-emulate is a method of fooling a guest OS (that is actually running on ring 1) into thinking that it is running in the kernel space on ring 0.

3.1 Issues with trap-and-emulate

- Some registers in the CPU reflect the actual privilege level. If the guest OS were to read these registers and detect that it is not running in kernel mode it might stop functioning normally.
- Some instructions that change system state run in both kernel and user space, but with different semantics. This might lead to the guest not trapping to the VMM in case of a privileged instruction being encountered.
- High performance overheads in processing interrupts.
- All ISAs do not support trap-and-emulate out of the box Most notably, Intel's x86 ISA did not support trap-and-emulate for a long time.

3.2 Issues with x86 virtualization

- The `popf` instruction is an example of an instruction that does not work with trap-and-emulate.
- In user mode, `popf` is used to change the ALU status flags. In kernel mode, `popf` is used to change system state flags (such as flags related to interrupt delivery).
- In user mode, no interrupt is generated by `popf` as it changes the system state. Hence even though the instruction is sensitive, it is not privileged as it does not issue a trap.
- There are 17 such instructions in the x86 ISA. Instructions like `pushf` reveal to the guest that it is running in user mode, while instructions like `popf` discussed above do not execute accurately.

3.3 Some definitions

3.3.1 Privileged

- State of the processor is privileged if
  - Access to that state breaks the virtual machine's isolation boundaries
  - It is needed by the monitor to implement virtualization

3.3.2 Strictly Virtualizable

- A processor or mode of a processor is strictly virtualizable if, when executed in a lesser privileged mode:
  - All instructions that access privileged state trap
  - All instructions either trap or execute identically

3.4 Binary Translation

- Binary translation is a method of implementing full virtualization. The steps involved in binary translation are:
  1. The VMM reads the next upcoming *basic block* of instructions. (By basic block we mean a logic block of instructions from the current point till the next branch)
  2. Each instruction in this basic block is translated to the target ISA, and the result is stored in a translation cache.
  3. Translation involves 3 types of instructions:
     - Instructions that can be directly translated and are safe (called **ident** instructions)
     - Short instructions that must be emulated using a sequence of safe instructions (eg: interrupt enable). This is called **inline** translation
     - Other dangerous instructions need to be performed by emulation code in the monitor. These are called **call-out** instructions. (eg: instructions that changes the PTBR).

3.5 Hardware-Assisted Virtualization

- The challenges of virtualizing x86 are outlined in section 3.2, and the methods to solve them were adopted as part of Intel's VT-x and AMD's AMD-V feature set
- The CPU now has 2 modes of operation, a **root** mode and a **non-root** mode.
- Both root and non-root mode have 4 rings. The current hardware state is maintained separately for both modes.
- The root mode is more powerful than the kernel mode. The host OS and VMM run in root mode, while the guest OS and applications run in non-root mode.
- If any sensitive instructions are executed in non-root mode, a VMEXIT condition signals to the processor to enter root mode. In root mode this sensitive operation is emulated by the VMM and the processor switches back to non-root mode.
- The hardware state of a VM is maintained in a data structure called the **Virtual Machine Control Structure** (VMCS). The VMM is in charge of creating the VMCS and modifying it (when emulating sensitive instructions).

4 Memory and I/O Virtualization

4.1 Memory Virtualization

- There are 3 address spaces that have to be translated for a successful memory reference by a guest OS. They are:
  1. The virtual address space of guest (also called **guest virtual address** or GVA)
  2. The physical address space of the guest (also called as the guest RAM or the **guest physical address** or GPA)
  3. The **host physical address** or the HVA
- The guest OS page table translates from GVA to GPA.
- The page tables maintained by the VMM translate from GPA to HPA (via the virtual address space of the host also called the HPA).

4.1.1 Shadow Page Tables

- The VMM creates a mapping from the GVA to the HPA (direct mapping) by combining the information available in the guest OS page table and the host OS page table.
- This direct mapping is called the shadow page table (SPT). It is offered to the hardware MMU (memory management unit) as a pointer to the base location. The pointer is stored in the control register `cr3`.
- While the guest is active, the VMM forces the processor to use the SPT for all translations.
- Whenever the guest OS modifies the guest page table, the VMM must update the shadow page table. This is implemented by making the guest page table *write protected*.
- This means that whenever the guest OS tries to write to the guest page table, a *page fault* is raised, and a trap is set to the VMM. The VMM handles the trap and modifies the SPT.

  For every guest application there is one shadow page table. Every time a guest application context switches, trap to VMM to change `cr3` to point to new shadow page table
- The drawbacks of the shadow page table concept is that it leads to overheads involved in handling traps, and the fact that the TLB cache has to be flushed on every context switch.

4.1.2 Extended Page Tables

- The processor is made aware of the virtualization, and the two-level address translation that is needed to support it.
- Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.
- A field in the VMCS maintains a pointer to the Extended page table, called the EPT Base Pointer.
- Benefits of EPT:
    1. Performance increased due to reduced overheads over shadow paging (performance increase is dependent on type of workload)
    2. Reduced memory footprint compared to SPT scheme that requires maintaining of a table for each VM that is started.

4.2 I/O Virtualization

- It is a technology that uses software to abstract upper-layer protocols from physical connections or physical transports.
- It involves managing and routing I/O requests from multiple virtual I/O devices and the shared physical hardware underneath.
- The three types of I/O virtualization are:
    - Full Device Emulation
    - Para I/O Virtualization
    - Direct I/O Virtualization

4.2.1 Full device emulation

- All functions of an I/O device (such as device enumeration and identification, interrrupt handling, DMA manipulation) are emulated entirely in software
- This software is located in the VMM and acts as a virtual device.
- The I/O access requests of the guest OS are trapped to the VMM which interacts with the I/O devices.

4.2.2 Para I/O Virtualization

- This is also called the **split driver model**. It consists of a front end driver that runs on the guest OS, and a back end driver that runs on the VMM.
- The front end and back end drivers interact with each other via shared memory.
- The front end driver intercepts I/O requests from the guest OS. The back end driver manages the physical I/O hardware as well as multiplexing the I/O data coming from different VMs
- Performance wise, para I/O virtualization is better than full device virtualization, but it comes with a high CPU overhead.

4.2.3 Direct I/O Virtualization

- Allows a guest to directly access the physical address of an I/O device. Virtual devices can directly perform DMA accesses to/from host memory.
- Intel VT-x technologies (if enabled) allow for a VM to directly write control information to a device's control registers. The VT-d extension allows for I/O devices to write into the memory that is controlled by VMs.

- The VMM utilizes and configures technologies such as Intel VT-x and Intel VT-d to perform address translation when sending data to and from an IO device.
- Advantage of faster performance, but limited scalability (as a single I/O device can only be assigned to a single VM).

4.2.4 Advantages of I/O Virtualization

- **Flexibility** due to abstraction of physical protocols, also leads to faster provisioning.
- **Minimization of costs** as there is now less need for hardware infrastructure like cables/switch ports/network cards.
- **Increased practical density** of I/O as it allows more connections to exist in a given space.

5 Goldberg and Popek Theorems

- Fundamental results postulated by R Goldberg and G Popek in 1975 that justify and prove that virtualization is possible to achieve.
- Goldberg and Popek classified the instructions in an ISA into the following categories:
    1. **Behaviour sensitive** instructions are those wherein the final result of the instruction is dependent on the privilege level (i.e. executing that instruction in a lower privilege level leads to a wrong output)
    2. **Control sensitive** instructions are those which result in change of processor state or processor privilege.
    3. **Privileged** instructions are those that trap if the processor is in user mode and do not trap if it is in system mode (i.e kernel or supervisor mode).

5.1 Requirements for virtualization support

As postulated by Goldberg and Popek, the requirements for an ISA to support virtualization are:

- **Equivalence**: A program executing on a VM must display essentially identical behaviour as one executing on an equivalent machine directly.
- **Resource Control**: A VM must be in total control of the virtualized resources
- **Efficiency**: A statistically dominant fraction of machine instructions must be executed without VMM intervention.

5.2 Theorems

5.2.1 Theorem 1

"For any conventional third generation computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions"

- The theorem states that to build a VMM it is sufficient that all instructions that could affect the correct functioning of the VMM (sensitive instructions) always trap and pass control to the VMM.

5.2.2 Theorem 2

"A conventional third generation computer is recursively virtualizable if it is:

1. virtualizable, and
2. A VMM without any timing dependencies can be constructed for it.

"

5.2.3 Theorem 3

"A hybrid virtual machine monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of the set of privileged instructions."

6 Live Migration of VMs

- Allows for real-time transfer of VMs from one physical node to another
- The challenge here is the design of a migration strategy that allows for migration but without affecting performance of the cluster of nodes.
- Why migration? Because of *load balancing*. Using the user login frequency and the load index, the most appropriate node for a given VM is chosen, to improve response time and increase resource utilization.
- Live migration is desired when load on the cluster becomes unbalanced and real-time correction is needed.
- Migration also allows for scalability (up and down) as well as rapid provisioning.

6.1 6-step migration process

6.1.1 Step 0 and 1: Migration Start

- Determining the source VM and the destination host.
- This can be started manually by a human user or by an automated load-balancing or server consolidation system.

6.1.2 Step 2: Iterative pre-copy

- Iteratively copy dirty pages from the source to the destination. Memory is copied page wise as it reflects the current execution state of the VM and it is required to continue the same functionality.
- This copy is carried out until the dirty portion of memory is small enough to be copied in a single final round.
- During the pre-copy phase, the functioning of the source VM is not interrupted.

6.1.3 Step 3: Stop and Copy

- The source VM is stopped, and the remaining memory state information is copied to the destination VM.
- During this phase the source VM's functioning is paused. This "downtime" must be made as short as possible.
- Non-memory state of the source VM such as the CPU state and network state is also sent in this step.

6.1.4 Step 4 and 5: Commitment and Activation

- The VM reloads the states and recovers the execution of programs in it, and the service provided by this VM continues.
- Then the network connection is redirected to the new VM and the dependency to the source host is cleared.
- The whole migration process finishes by removing the original VM from the source host.

6.2 Pre-copy and post-copy migration

- In **pre-copy migration**, the aim is not to impact the functioning of the source VM. However since the migration daemon is making use of the network to transfer dirty pages, there is a degradation of performance that occurs.
- Adaptive rate-limited migration is used to mitigate this to an extent.
- Moreover, the maximum number of iterations must be set because not all applications' dirty pages are ensured to converge to a small writable working set over multiple rounds.
- In **post-copy migration**, the migration is initiated by stopping the source VM, a minimal subset of the execution state of the VM is transferred to the target. The VM is then resumed at the target.
- Concurrently, the source actively pushes the remaining memory pages of the VM to the target - an activity known as pre-paging.
- At the target, if the VM tries to access a page that has not yet been transferred, it generates a page-fault. These page faults are trapped, sent to the source and the source replies with the page requested.

7 Lightweight Virtualization

7.1 Containers

- Containers are a logical packaging mechanism where the code and all of its dependencies are abstracted away from their run time environment.
- This allows for much easier deployment on a wide variety of hardware, as well as more effective isolation and much less CPU/Memory overheads.
- Containers are an example of *OS-level virtualization*, and multiple containers running on a host share the same OS. Similar to a VMM for full-scale virtual machines, containers are managed by a container manager.
- Examples of real world implementation of container technology are Docker, Google's Kubernetes Engine, AWS Fargate, Microsoft Azure etc.

7.2 Docker

- Docker is a product that is used to deliver software in the form of containers, and it makes use of Linux technologies that promote OS-level virtualization such as **cgroups**, **namespaces** and others.
- Docker consists of 3 components:
  1. The **Docker engine**
  2. The **Docker client** (normally a command line interface which is called the Docker CLI)
  3. The **container registry**
- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- The container registry stores Docker images. An example of a publicly-available registry is **Docker Hub**. By default, the `docker pull` and `docker run` commands pull the needed images from Docker Hub.
- It is possible to configure Docker to look elsewhere for images, including one's own privately set up registry.
- The **Docker Engine** is a client-server program. The Docker CLI acts as an client and uses the Docker API to send requests. The engine listens for these requests, and sends them to the Docker daemon running on the server.

7.2.1 Docker Images

- A Docker image is a read-only template that is used to set up a running container.
- It provides a convenient way to package up applications and pre-configured server environments, which can be used for private use or to share publicly with other Docker users.
- Each of the files that make up a Docker Image is called a *layer*. Layers are treated by Docker as intermediate images that are built in a specific order (each layer being dependent on the layers below it)
- Layers that change the most often are organized at the top, as then there are minimal number of layers that need to be rebuilt each time a change occurs (when a layer changes only the layers above it must be rebuilt).
- When a container is launched from an image, a thin writable layer called the *container layer* is added at the top. The container layer stores all the changes made to the container state as it runs.
- This allows for multiple containers to share the same image layers but only have their distinct container layers at the top.
- A **Dockerfile** is a plain-text file that specifies the steps involved in creating a Docker image.

7.3 Linux namespaces

- A namespace is a method of partitioning processes into groups such that different groups see different sets of resources.
- The resources that are partitioned in this way can be the file system, networking, interprocess communication as well as process IDs.

7.3.1 Mount namespace

- Mount namespaces allow a process that lies within a namespace to have a completely different view of the system mount structure from the actual one.
- This promotes isolation as it allows each isolated process to have its own separate file system root, hence it avoids exposing more information about the overall file system than is needed.
- Any mount/unmount operations that are done by the isolated process in its own mount namespace will not affect the parent mount namespace, nor any other isolated mount namespace in the hierarchy.

7.3.2 UTS namespace

- UTS (UNIX Time-Sharing) namespaces allow a single system to appear to have different host and domain names to different processes.
- When a process creates a new UTS namespace, the hostname and domain of the new UTS namespace are copied from the corresponding values in the caller's UTS namespace

7.3.3 Network namespace

- Network namespaces allow processes to see entirely different sets of network interfaces.
- Each network namespace consists of the following objects:
  - Network devices (labelled as `veth` or Virtual Ethernet devices)
  - Bridge networks
  - Routing tables
  - IP Addresses
  - Ports

- Virtual network interfaces span multiple network namespaces, and allow interfaces in different namespaces to communicate with one another.
- A routing process takes data incoming at the physical interface, and routes it to the correct network namespace via the virtual network interface
- Routing tables can be set up that route packets between virtual interfaces.

7.3.4 Linux Control Groups (cgroups)

- Developed by Paul Menage, Rohit Seth and others at Google (2006), it is a Linux kernel feature that allows the limiting, accounting and isolation of resources (CPU, memory, disk, I/O, network etc.) for a group of processes.
- Functionality of cgroups is as follows:
  - Access: which devices can be accessed by a particular cgroup
  - Resource limiting
  - Resource prioritization (between cgroups)
  - Accounting
  - Control: freezing processes and checkpointing

7.4 Container File System: UnionFS

- The drawbacks of existing file systems w.r.t. containerized services are:
  - **Inefficient disk-space utilization**: If 10 instances of a single Docker container (each of size 1 GB) are started, then on a traditional FS totally it takes up 10 GB of memory.
  - **Latency in startup**: Given that a container is essentially a process, the only way for a process to be created is a `fork()` system call. The inefficiency is because each time a container has to be started, all the image layers have to be copied into the new process address space, which takes time for large number of image layers.
- UnionFS is a unified and coherent view to files in separate file systems. It allows for multiple file systems to be mounted onto a single root.
- It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.
- Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual file system.
- This allows a file system to appear as writable, but without actually allowing writes to change the file system, also known as copy-on-write.
- In the CoW mechanism, any changes that are made to any of the image layers that make up the UnionFS, are reflected only in the topmost container layer. The image layer is *copied* to the container layer FS and changes are written there.
- Refer to this link

7.4.1 Disadvantages

- Translating between different file system rules about file names and attributes, as well as different file system's features.
- Copy-on-write makes memory-mapped file implementation hard
- Not appropriate for working with long-lived data or sharing data between containers, or a container and the host.

8 DevOps on the cloud

- **DevOps** is an integration of Software Development methodologies and IT operations that are involved in deployment and operation of software.
- DevOps auomates the process that occur between software development and IT teams so that software can be built, released and tested faster and more reliably.
- One of the key principles of DevOps is **Continuous Integration** along with **Continuous Deployment** and **Continuous Delivery** (this is commonly referred to as **CI\CD**).
- CI\CD promotes the practice of making small changes and integrating them with the main codebase often, and using automated deployment infrastructure to test on a production-like environment.
- The entire CI\CD sequence of stages is organized in the form of a sequential *pipeline*. The pipeline consists of a series of automated actions that take code from a developer environment to a production environment.
- Pipelines automate the build, test and publishing of artifacts so that they can be deployed to a runtime environment.
- Tools such as **Jenkins**, **Drone**, and Travis CI are used for CI\CD pipeline management.
- A typical CI\CD pipeline is as follows:
  - Developer push their changes to a centralized Git repository
  - Build server automatically builds the application and runs unit tests and integration tests on it
  - If all tests pass then container image is pushed to the central container repository.
  - The newly built container is automatically deployed to a staging environment
  - The acceptance tests are carried out in this staging environment.
  - Verified and tested container image is pushed to production environment.

8.1 Continuous Integration (CI)

- CI is a development practice wherein developers integrate code into a shared codebase (implemented as a repository) at a high frequency (maybe several times a day).
- Each integration can then be verified by an automated build and automated tests.
- CI consists of the following workflows:
  1. Development and unit testing in the developer's local environment
  2. Compile code on automated build server
  3. Run additional static analyses, measure and profile performance, generate documentation and facilitate manual QA processes
  4. Integration with Continuous Delivery (make sure code is always at a deployable state) and Continuous Deployment (automate deployment).

8.2 Continuous Deployment (CD)

- Automated deployment of successful builds to the production environment.
- In an environment in which data-centric microservices provide the functionality, and where the microservices can be multiply instantiated, CD consists of instantiating the new version of a microservice and retiring the old version.

8.3 Jenkins

- A self-contained, open source automation server which can be used to automate tasks related to building, testing, and delivering or deploying software.
- It can be installed via package managers (`apt-get`, `pacman`), DockerHub, or natively built on a machine with Java Runtime Envt (JRE).
- Plugins are used to extend Jenkins functionality as per the user-specific or organization-specific needs
- Some commonly used Jenkins plugins are:
  - Dashboard view, view job filters
  - Monitoring and metrics
  - Kubernetes plugin
  - Build pipeline
  - Git and GitHub integration

9 Container Orchestration and Kubernetes

- The process of deploying containers on a compute cluster consisting of multiple nodes.
- Includes managing container lifecycles in large and dynamic environments (especially in microservice architectures where each microservice is implemented as a container)
- Orchestrator is a software that virtualizes different physical nodes into a single compute infrastructure for the user to deploy containers on.
- It automates deployment, scaling, management, networking and availability of container-based apps.
- Scheduling: managing the resources available and assigning workloads where they can most efficiently be run.
- Cluster management: joining multiple physical or virtual servers into a unified, reliable, fault-tolerant group.
- Typically orchestrators take care of all 3: orchestration, scheduling and cluster mgmt.
- Kubernetes (or K8s for short) is the most prominent example of such a software. Others are Docker Swarm, Google Container Engine (built on Kubernetes), and Amazon ECS.

9.1 K8s Architecture

9.1.1 K8s Pod

- K8s manages applications that consist of communicating microservices
- Often those microservices are tightly coupled forming a group of containers that would typically, in a non-containerized setup run together on one server.
- This smallest unit that can be scheduled to deploy on K8s is called a **pod**.
- The containers in a pod share cgroups, namespaces, storage and IP Addresses as they are co-located.
- Pods have a short lifetime, they are created, destroyed and restarted on demand.

9.1.2 K8s Service

- As pods are shortlived, there is no guarantee on their IP Address, which makes communication hard
- A service is an abstraction on top of a number of pods, typically requiring to run a proxy on top, for other services to communicate with it via a Virtual IP address.
- Numerous pods can be exposed as a service with configured load balancing.

9.1.3 Master Node

- Consists of an API Server, a key-value store called `etcd`, scheduler and controller-manager
- The **API server** serves REST API requests according to the bound business logic.
- `etcd` is a consistent and simple key-value store that is used for service discovery and shared config storage. It allows for CRUD operations and notification services to notify the cluster about config changes.
- **Scheduler** deploys configured pods and services onto the worker nodes. It decides based on the resources available on each cluster.
- **Controller-manager** is a daemon that enables the use of various control services. It makes use of the API server to watch the current state and make changes to the config to maintain the desired state (e.g.: maintaining the replication factor by reviving any dead/failed pods)

9.1.4 Worker Node

- Consists of Docker, `kubelet`, `kube-proxy`, and `kubectl`
- **Docker** runs the configured pods, takes care of downloading the images and starting the containers.
- `kubelet` gets the configuration of a pod from the API Server and ensures that the described containers are up and running. It also communicates with `etcd` to read and write details on running services.
- `kube-proxy` is a network proxy and load balancer for a single node that routes TCP and UDP traffic
- `kubectl` is a command line tool that sends API requests to the API server.