CHAPTER TWO

LITERATURE REVIEW

2.1 Preamble

In today's modern world, we are experiencing advancement in technologies and many electronic devices which offer tremendous performance irrespective of size. These new technologies already form a very large part of everyday activities. Such consumer electronics are desktops, laptops, tablets, smart phones, and so on. simply referred to as digital electronic. They are also used in wide variety of disciplines or domains such as automotive industry, aerospace engineering (avionics and aeronautic engineering), and medicine, among others. The constant demand for high performance has resulted in a rapid development of semi-conductor technologies miniaturizing computing devices with the assistance of very large number of transistors, the fundamental building block of modern electronic devices in a chip. This is in line with Moore's Law even though experts agree that computers should reach the physical limits of this law at some point in the 2020s. The reason being that high temperatures of transistors eventually would make it impossible to create smaller circuits because cooling down the transistors takes more energy than the amount of energy that already passes through the transistors. There are therefore, enormous gains in technology scaling. However, this comes at a cost that devices manufactured in the latest technologies may be affected by errors which halt the flow of processes or tasks.

Increasing soft error rates in recent semiconductor technologies enforce the usage of fault tolerance. While fault tolerance enables correct operation in the presence of soft errors, it usually introduces a time overhead. The time overhead is particularly important for a group of computer systems, especially in the cloud computing, referred to as real-time systems (RTSs) where correct operation is defined as producing the correct result of a computation while satisfying given time

constraints (that is, deadlines). Depending on the consequences when the deadlines are violated, RTSs are classified into soft and hard real-time systems.

2.2 Concepts of fault-tolerant computing

Fault tolerance simply means the ability of a system to continue operating uninterrupted despite the failure of one or more of its components. This is true whether it is a computer system, a cloud cluster, a network, or something else The early work of Huang and Abraham (1984), first plied into the concept of fault tolerance. Implicit in the definition of fault tolerance is the assumption that there is a specification of what constitutes correct behavior. A failure occurs when an actual running system deviates from this specified behavior. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behavior specification. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures. For example, in a software system, an incorrectly written instruction in a program may decrement an internal variable instead of incrementing it. Clearly, if this statement is executed, it will result in the incorrect value being written. If other program statements then use this value, the whole system will deviate from its desired behavior. In this case, the erroneous statement is the fault, the invalid value is the error, and the failure is the behavior that results from the error. If the variable is never read after being written, no failure will occur. Or, if the invalid statement is never executed, the fault will not lead to an error. Thus, the mere presence of errors or faults does not necessarily imply system failure. As this example illustrates, the designation of what constitutes a fault - the underlying cause of a failure - is relative in the sense that it is simply a point beyond that anyone may not want to delve further. After all, the incorrect statement itself is really an error that arose in the process of writing the software, and

so on. At the heart of all fault tolerance techniques is some form of masking redundancy. This means that components that are prone to defects are replicated in such a way that if a component fails, one or more of the non-failed replicas will continue to provide service with no appreciable disruption.

2.2.1 Fault tolerance

In principle, fault tolerance is based on redundancy and this takes various forms: Information redundancy. Adding redundant information helps detecting, or even correcting errors in storage or in transmission. Temporal redundancy. This covers the use of replicated processing: the same action is replicated in several instances in parallel to increase the probability that it will be achieved at least once in spite of failures. Spatial redundancy. This covers the use of replicated data: information is maintained in several copies to reduce the probability of data loss or corruption in the presence of failures.

Cloud computing is a style of computing where services are provided across the internet using different models and layers of abstraction. It refers to the applications delivered as services to the masses, ranging from the end-users hosting their personal documents on the internet to enterprises outsourcing their entire information technology (IT) infrastructure to external datacentres. Some simple examples of cloud computing service are Google drive, One drive, Yahoomail, Gmail, and the likes. Due to the exponential and fast growth of cloud computing, the need for fault tolerance in the cloud is a key factor for consideration. Fault tolerance is concerned with all the techniques necessary to enable a system to tolerate software or hardware faults remaining in the system after its development.

Fault tolerance enables a system to continue to perform its operations, possibly at a reduced level, rather than failing completely, when some subcomponents of the system malfunctions

unexpectedly. Fault tolerance is an important aspect in cloud storage, due to the strength of the stored data (Anju and Inderveer 2012). The main benefits of implementing fault tolerance in cloud computing include failure recovery, the simple hardware platform needed, being independent from application, high reliability and availability, cost effectiveness, improved performance metrics among others.

Fault attacks have been inevitable and become a serious concern in the cloud computing industry where resources are provisioned. In everyday language, the terms fault, failure and error are used interchangeably. In fault-tolerant computing parlance, however, they have distinctive meanings. A fault (or failure) can be either a hardware defect or a software/programming mistake (bug). In contrast, an error is a manifestation of these fault or failure and bug. The general path of failure is given in figure 2.1

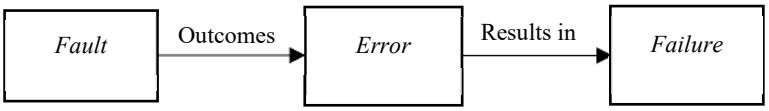

Figure 2.1: General path of failure

Fault tolerant systems provide a clear exposition of these attacks and the protection strategies that can be used to thwart or recover from them whenever they occur. Fault tolerance is the property that enables a system to continue with its correct operation even in the presence of faults (errors), and it is generally implemented by error detection and subsequent system recovery (Koren and Krishna, 2010). Fault tolerance has been a subject of research for a long time and significant amount of work has been produced over the years (Al-Omari, *et al.,* 2004). To provide fault tolerance, systems are usually designed such that some redundancy is included. The common types of redundancy used are information, hardware and time redundancy.

Error-detecting and error-correcting codes provide fault tolerance while using information redundancy, that is, the data includes additional information (check bits) that can verify the

correctness of the data before it is used (error-detection), or even correct erroneous data bits (error-correction). Different error-detecting and error-correcting codes have been proposed including parity codes, cyclic codes, arithmetic codes, hamming codes, and so on. (Huffman and Pless, 2003). The major disadvantage of error-detecting and error-correcting codes is that they are limited to errors that occur during transfer of data (system bus) or errors in memory in the case of information or software error.

Employing hardware redundancy is a very common practice to provide fault tolerance. Already in 1952, John von Neumann introduced a redundancy technique called NAND multiplexing for constructing reliable computation from unreliable devices (Von Neumann,1956). *n*-Modular Redundancy (NMR) is another method that provides fault tolerance at the cost of adding *n*-hardware replicas. The correct output in an NMR is obtained by voting on the outputs generated from the *n* modules. The most common form of NMR is Triple Modular Redundancy (TMR) (Kastensmidt, *et al.,* 2005). In TMR, a process is executed on three functionally equivalent hardware modules and the output is obtained by using the two-out-of-three voting concept. Thus, even if one of the hardware units in TMR fails, the system is still able to produce the correct output. Multiple variations of NMR exist. Unit-level modular redundancy applies the same concept as NMR at a higher granularity, that is, instead of only replicating entire modules, it replicates also units (subsystems) within the modules. Dynamic redundancy is another variation of NMR, but the major difference is that even though there are N replicated modules, only one module is active at a time, and in case of errors, the active module is replaced by a spare module. Hybrid redundancy requires even more hard- ware redundancy because it assumes that all N module are active at a time, and in case errors are detected in some modules, the erroneous modules are replaced by spare modules. In hybrid redundancy, the erroneous modules are detected

by comparing the output of the voter with the outputs of the active modules, and all the active modules which are in disagreement with the voter need to be replaced by spare modules. Sift-out modular redundancy is another variation of NMR, but instead of using a majority voter, this technique uses comparator, detector, and collector circuits, where the comparator and the detector circuits are used to exclude the erroneous modules from the system, and the collector circuit is used to provide the system output (Kastensmidt, *et al.,* 2005).

The major advantage of NMR, and any of the similar techniques discussed earlier, is that it is capable of error-masking (disabling propagation of errors), as long as less than half of the modules are affected by errors. Still, if more than half of the modules are affected by errors, NMR is able to detect errors, and in such case a system recovery needs to be performed. The major disadvantage of NMR is that it is a costly solution to implement fault tolerance because it requires substantial amount of hardware redundancy.

Time redundancy is another way to provide fault tolerance. However, fault tolerance techniques that use time redundancy are only efficient if the faults are of transient nature, that is, faults that occur, but disappear after a short period of time. These transient faults often result in soft errors. The simplest technique that uses time redundancy copes with soft errors by executing the same program twice, and it obtains the correct result if the outputs of the two executions match. Rollback Recovery with Checkpointing (RRC) is a well-known fault tolerance technique that efficiently copes with soft errors. RRC has been the focus of research for a long time (Ling *et al.*, 2001). Unlike classical re-execution schemes where the task (job) is restarted once an error is detected, RRC copes with soft errors by making use of previously stored error-free states of the task, referred to as checkpoints. During the execution of a task, the task is interrupted and a checkpoint is taken and stored in a memory. The checkpoint contains enough information such

that a task can easily resume its execution from that particular point. For RRC it is crucial that each checkpoint is error-free, and this can be done by for example running acceptance tests to validate the correctness of the checkpoint. Once the checkpoint is stored in memory, the task continues with its execution. As soft errors may occur at any time during the execution of a task, an error detection mechanism is used to detect the presence of soft errors. There are various error detection mechanisms that can be used, for instance, watchdogs, duplication schemes and so on. (Kwak *et al.,* 2012). In case that the error detection mechanism detects an error, it forces the task to roll-back to the latest checkpoint that has been stored. A new and modern-day method is the introduction of VMs.

2.2.2 Why fault tolerance?

The aim of fault tolerant design in cloud computing majorly is to minimize the probability of failures, whether those failures simply annoy the users or result in lost fortunes, human injury, or environmental disaster.

i. It is practically impossible to build a perfect system: As the complexity of a system grows, its reliability drastically decreases, unless compensatory measures are taken.
ii. Faults are likely to be caused by situations outside the control of the designers. Such situations include environmental factors and unforeseen mistakes by potential users.

2.3 Taxonomy of fault tolerance

Today, fault tolerance has become more important as an integral part of cloud computing. It has appeared as an acceptable and indispensable model where the processing of very large volume of data is involved in a distributed computing environment. Fault tolerance is an unavoidable trend in the present computing in general and cloud computing in particular where resources are provision with service level agreement. Figure 2.2 gives the taxonomy of fault tolerance and its relationship with cloud computing.

In fault tolerance, there are types and categories of faults. Faults can be classified on several factors such as:

i. Network fault: A Fault occur in a network due to network partition, packet loss, packet corruption, destination failure, link failure, and so on.
ii. Physical fault: This Fault can occur in hardware like fault in CPUs, memory, storage media.
iii. Processor fault: Fault occurs in processor due to operating system crashes, and so on.
iv. Process fault: A fault which occurs due to shortage of resource, software bugs, interrupts, and so on.
v. Service expiry fault: The service time of a resource may expire while application is using it.

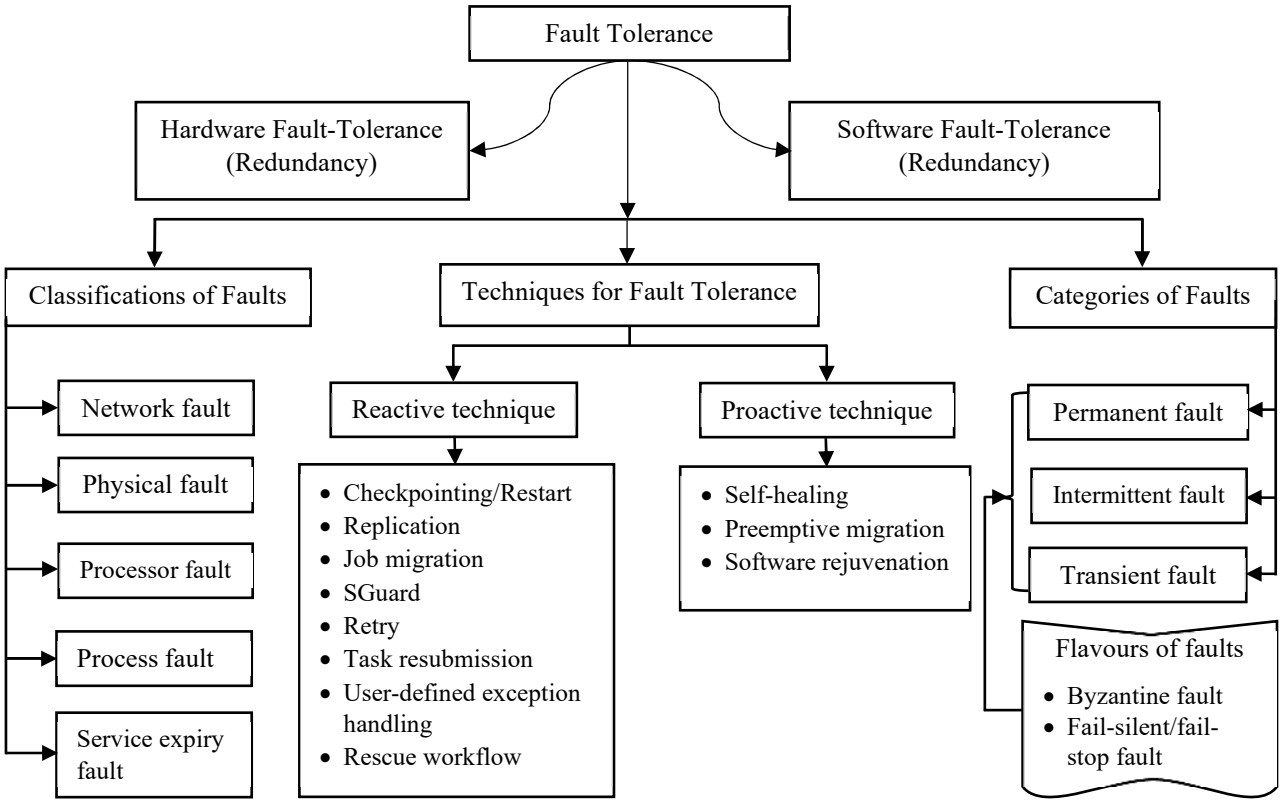

Figure 2.2: Taxonomy of Fault Tolerance

2.3.1 Categories of fault

There are three main categories or types of faults: permanent, intermittent, and transient. Figure 2.3 conceptualizes these categories of faults. A fault can be categorized on the basis of computing resources and time. A failure occurs during computation on system resources can be classified as: omission, timing, response and crash failure.

i. Transient: A transient fault is a fault that happens once, and then does not ever happen again depicted in figure 2.3(a). For example, a fault in the network might result in a request that is being sent from one node to another to time out or fail. These failures are caused by some inherent fault in the system. However, these failures are corrected by retrying roll back the system to previous state such as restarting software or resending a message. These failures are very common in computer systems.

ii. Intermittent: An intermittent fault is one that occurs once, seems to go away, and then occurs again as shown in Figure 2.3(b). They are sometimes the hardest ones to debug and deal with, since they masquerade as transient faults at first, but then come back, sometimes with inconsistency. A good example of this is with loose connections in hardware, where sometimes it seems like the connection works, but occasionally (and often erratically), the connection just stops working for a bit. Mostly these failures are ignored while testing the system and only appear when the system goes into operation. Therefore, it is hard to predict the extent of damage these failures can bring to the system.

iii. Permanent: A permanent fault is one that just does not go away after it first occurs. A permanent fault occurs once, and then continues to persist until it has been addressed. See Figure 2.3(c). For example, if part of a system runs out of memory, hits an infinite loop, or crashes unexpectedly, that "broken" state will continue to be the same until the fault is fixed or replaced. These failures can cause major disruptions and some part of the system may not be functioning as desired.

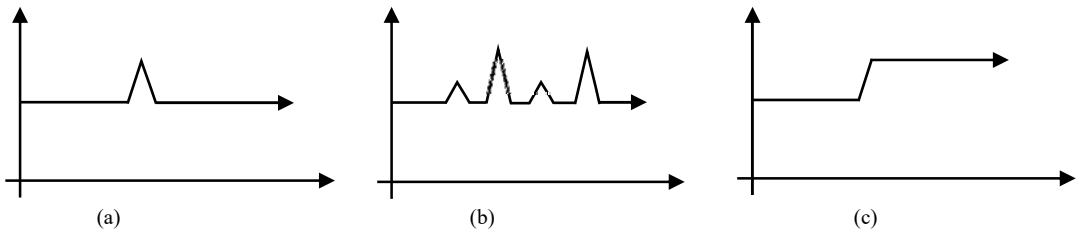

Figure 2.3: Conceptualization of transient, intermittent and permanent faults.

2.3.2 Flavours of faults

There are two main flavours of faults and each of these three categories of faults discussed above comes under these flavours. Every transient, intermittent or permanent fault runs the risk of either being a fail-silent or a byzantine fault.

1. **Fail-silent (or fail-stop) fault**: This is one where the node or VM where the fault originated stops working. Here, when the origin VM stops working, it will either produce no result (error/output) whatsoever or produce some sort of output that indicates that the VM actually failed. In a fail-silent fault, there is no guarantee that the VM with the fault will actually give an error, so it is possible that that the occurrence of fault is unnoticed.
2. **Byzantine fault**: Here, the origin VM produces an error output but does not always produce the same error output. Confusingly, even though the VM is producing errors, it continues to run. In a Byzantine fault, a VM could behave inconsistently in the exact errors that it surfaces, which means that a single fault within a VM could actually result in the VM responding with various different errors, all of which are different from one another.

Both fail-silent (fail-stop) and byzantine faults are definitely treacherous scenarios. One of the aims of this research is to build a model or system for both situations even though it is important to note that a fault-free fault-tolerant system cannot be completely built (though it is good to try to strive in this direction).

2.3.3 Techniques for fault tolerance

There are two major techniques for fault tolerance which are reactive and proactive techniques. These techniques are sub-classified base on certain policies and their modes of operations. Table 2.1 shows the tools used to implement some of these fault tolerance techniques.

i. **Reactive Fault Tolerance**: Reactive fault tolerance policies reduce the effect of failures on application execution when the failure effectively occurs. Various techniques based on this policy are:

ii. **Checkpointing/Restart**: When a task fails, it is allowed to be restarted from recently checked pointed state rather than from the beginning. In this scenario after doing every change It is an efficient task level fault tolerance for long running and big applications. In this scenario after doing every change a checkpointing is done.

iii. **Replication**: Replication means copy. Various tasks are replicated and they are run on different resources, for the successful execution and for getting the desired result. It can be implemented using tools like Hadoop and AmazonEc2 and so on. In order to make the execution succeed, various replicas of task are run on different resources until the whole replicated task is not crashed. HAProxy (High Availability Proxy), Hadoop and AmazonEC2 are used for implementing replication. Using tools like HA-Proxy, Hadoop and AmazonEC2 replication can be implemented.

iv. **Job Migration**: On the occurrence of failure, the job is migrated to a new machine. HA Proxy can be used for migrating the jobs to other machines. Some time it happened that due to some reason a job can- not be completely executed on a particular machine. At the time of failure of any task, task can be migrated to another machine. Using HA-Proxy job migration can be implemented.

v. **S-Guard**: It is less turbulent to normal stream processing. S-Guard is based on rollback recovery and can be executed in HADOOP (an open-source software (distributed processing) framework that manages data processing and storage for big data applications running in clustered systems), Amazon EC2 (Elastic Compute Cloud).

vi. **Retry**: It is the simplest technique that retries the failed task on the same resource. The user resubmits the task on the same cloud resource. In this case a task is implemented again and again.

vii. **Task Resubmission**: A job may fail now. In this case at runtime the task is resubmitted again either to the same machine on which it was operating or to some other machine.

viii. **User defined exception handling**: Here the user defines the specification of a task failure for workflows.

ix. **Rescue workflow**: It allows the system to keep functioning after failure of any task until it will not be able to proceed without rectifying the fault.

2. Proactive Fault Tolerance: This refers to avoiding failures, errors and faults by predicting them in advance and replace the suspected components by other working components thus avoiding recovery from faults and errors (Amin *et al.,* 2015). Some of the techniques which are based on these policies are:

i. **Self-healing**: When multiple instances of an application are running on multiple VMs, it automatically handles failure of application instances. A big task can be divided into parts. This Multiplication is done for better performance. When various instances of an application are running on various VMs, it automatically handles failure of application instances.

ii. **Preemptive Migration**: In this technique an application is constantly observed and analyzed. Preemptive Migration relies on a feedback-loop control mechanism where application is constantly monitored and analyzed.

iii. **Software Rejuvenation**: The system is planned for periodic reboots and every time the system starts with a new state.

Table 2.1: Tools Used to Implement Existing Fault Tolerance Techniques; Source: (Bala & Chana, 2012)

Fault Tolerance Technique	System	Programming Framework	Environment	Fault Detected	Application Type
Self -Healing, Job Migration, Replicatio	HAProxy (Yadav & Pandey, 2012)	Java	Virtual Machine	Process/Node failures	Load Balancing
Checkpointing	SHelp (Gang *et al*., 2010)	SQL, JAVA	Virtual Machine	Application Failure	Fault Tolerance
Checkpointing, Retry, Self -Healing	Assure (Sidiroglou *et al.* 2009)	JAVA	Virtual Machine	Host, Network Failure	Fault Tolerance
Job Migration, Replication, S-guard,	Hadoop (Shankar & Ravi, 2014)	JAVA, HTML, CSS	Cloud Environment	Application/Node Failure	Data intensive
Replication, Task Resubmission	Amazon EC (Kulkarni, Sutar, & Gambhir, 2012)	Amazon Machine Image, Amazon Map	Cloud Environment	Application/Node Failure	Load Balancing

2.4 Real-time systems

The term real-time is derived from its use in early simulation, in which a real-world process is simulated at a rate that matched that of the real process. Analog computers, most often, were capable of simulating at a much faster pace than real-time, a situation that could be just as dangerous as a slow simulation if it were not also recognized and accounted for. In computer science, real-time computing (RTC), or reactive computing describes hardware and software systems subject to a "real-time constraint", for example from event to system response. Real-time programs must guarantee response within specified time constraints, often referred to as "deadlines". The correctness of these types of systems depends on their temporal aspects as well as their functional aspects. Real-time responses are often understood to be in the order of milliseconds, and sometimes microseconds. A system not specified as operating in real time cannot usually guarantee a response within any timeframe, although typical or expected response times may be given.

A real-time system has been described as one which "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" (Krishna, 2010). The term "real-time" is also used in simulation to mean that the simulation's clock runs at the same speed as a real clock, and in process control and enterprise systems to mean "without significant delay".

The concept of real-time systems is represented in figure 2.4. RTSs span several domains of computer science. They are in defense and space systems, networked multimedia systems, avionics, embedded electronics and so on. In a real-time system the correctness of the system behavior depends not only the logical results of the computations, but also on the physical instant at which these results are produced. A real-time system changes its state as a function of physical time, for instance, a chemical reaction continues to change its state even after its controlling computer system has stopped. Based on this, a real-time system can be decomposed into a set of subsystems that is, the controlled object, the real-time computer system and the human operator. A real-time computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment. The instant at which a result is produced is called a deadline. If the result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is firm. If a catastrophe could result when a firm deadline is missed, the deadline is hard. Commands and Control systems, Air traffic control systems are examples for hard real-time systems. Online transaction systems, airline reservation systems are soft real-time systems.

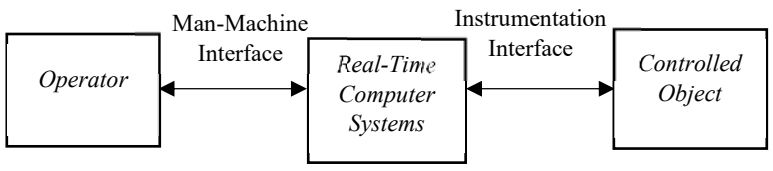

Figure 2.4: The Real-Time System

2.4.1 Orthogonality of fault tolerance to real-time systems

Fault tolerance is defined informally as the ability of a system to deliver the expected service even in the presence of faults. A common misconception about real-time computing is that fault-tolerance is orthogonal to real-time requirements. It is often assumed that the availability and reliability requirements of a system can be addressed independent of its timing constraints. This assumption, however, does not consider the distinction between the characteristic of real-time computing that is, the correctness of a system being dependent not only on the correctness of its result, but also on meeting stringent timing requirements. In other words, a real-time system may fail to function correctly either because of errors in its hardware and/or software or because of not responding in time to meet the timing requirements that are usually influenced by its environment. Hence, a real-time system can be viewed as one that must deliver the expected service in a timely manner even in the presence of faults. A missed deadline can be potentially as disastrous as a system crash or an in correct behavior of a critical task for example, a digital control system may lose stability.

In fact, if the logical correctness of a system may be dependent on the timing correctness of certain components separating the functional specification from the timing specification is a very difficult task. Moreover, timeliness and fault-tolerance could sometimes pull each other into opposite directions. For example, frequent extra checks and exotic error recovery routines will enhance fault-tolerance but may increase the chance of missing the deadlines of application tasks.

When a system specification requires certain service in a timely manner, then the inability of the system to meet the specified timing constraint can be viewed as a failure. However, a simple approach of applying existing fault-tolerant system design methods by treating a missed deadline as a timing fault does not address fully the needs of real-time applications. The fundamental

difference is that real-time systems must be predictable, even in the presence of faults. Hence, fault-tolerance and real-time requirements must be considered jointly and simultaneously when designing such systems (Sun *et al.,* 2005). The challenge is to include the timing and the fault-tolerance requirements in the specification of the system at every level of abstraction and to adopt a design methodology that considers system predictability even during fault detection, isolation, system reconfiguration and recovery phases.

2.4.2 Classification of real-time systems

Kopetz (2011), classified real-time systems in different perspectives. The first two classifications, hard real-time versus soft real-time and fail-safe versus fail-operational, depend on the characteristics of the application, that is, on factors outside the computer system. The second three classifications, guaranteed-timeliness versus best-effort, resource-adequate versus resource-inadequate, and event-triggered versus time-triggered, depend on the design and implementation, that is, on factors inside the computer system.

2.4.3 Hard real-time versus Soft real-time

Table 2.2 shows the major differences between hard and soft real-time systems. The response time requirements of hard real-time systems are in the order of milliseconds or less and can result in a catastrophe if not met. In contrast, the response time requirements of soft real-time systems are higher and not very stringent. In a hard real-time system, the peak-load performance must be predictable and should not violate the predefined deadlines. In a soft real-time system, a degraded operation in a rarely occurring peak load can be tolerated. A hard real-time system must remain synchronous with the state of the environment in all cases. On the other hand, soft real-time systems will slow down their response time if the load is very high. Hard real-time systems are often safety critical. Hard real-time systems have small data files and real-time databases. Temporal accuracy is often the concern here. Soft real-time systems for example, on-line

reservation systems have larger databases and require long-term integrity of real-time systems. If an error occurs in a soft real-time system, the computation is rolled back to a previously established checkpoint to initiate a recovery action. In hard real-time systems, roll-back/recovery is of limited use.

Table 2.2: Hard real-time versus soft real-time systems

S/N	Characteristic	Hard Real-Time	Soft Real-Time
1	Response time	Hard-required	Soft-desired
2	Peak-load performance	Predictable	Degraded
3	Control of pace	Environment	Computer
4	Safety	Often critical	Non-critical
5	Size of data files	Small/medium	Large
6	Redundancy type	Active	Checkpoint–recovery
7	Data integrity	Short-term	Long-term
8	Error detection	Autonomous	User assisted

2.4.4 Real-time scheduling

A hard real-time system must execute a set of concurrent real-time tasks in a such a way that all time-critical tasks meet their specified deadlines. Every task needs computational and data resources to complete the job. The scheduling problem is concerned with the allocation of the resources to satisfy the timing constraints. Hard real-time scheduling can be broadly classified into two types: static and dynamic. In static scheduling, the scheduling decisions are made at compile time. A run-time schedule is generated off-line based on the prior knowledge of task-set parameters, for instance, maximum execution times, precedence constraints, mutual exclusion constraints, and deadlines.

2.5 Cloud computing

Cloud computing sets a new paradigm for infrastructural management by offering unprecedented possibilities to deploy software in cloud computing environments (Armbrust *et al.,* 2010). Cloud computing is a model for enabling ubiquitous, convenient and on-demand network access to a shared pool of configurable computing resources. Such resources are networks, servers, storage,

applications and even services. They can be rapidly provisioned and released with little management effort or service provider interaction. The goal of cloud computing is to share resources among the cloud users/consumers, cloud partners and vendors in the cloud value chain (Vecchiolaa *et al.,* 2012). Amazon's Elastic compute cloud (EC2) is one of the most well-known cloud utility providers with model that enables on-demand provisioning of computational resources using VMs in cloud providers' datacentres (Armbrust *et al.,* 2010). The basic theory of cloud computing is that IT resources are made available within an environment that enables them to be used, via a communications network as a service (Balco, Law, & Drahošová, 2017). Several researchers have investigated and come up with various definitions for Cloud computing. Some of these definitions are shown in Table 2.3.

While cloud computing has brought about a transformation in the delivery model of information technology from a product to a service, and it has also enabled the availability of various software, platforms and infrastructural resources as scalable services on demand over the internet, the performance of cloud computing services is hampered due to their inherent vulnerability to failures owing to the scale at which they operate (Kumari and Kaur, 2018).

The basic theory of cloud computing is that resources are made available within an environment that enables them to be used via a communication network as a service (Balco, *et al.*, 2017). More businesses are implementing platforms that provide functionality for performing commercial transactions over the cloud. With the exponential growth of cloud computing as a solution for providing flexible resources, dynamically detecting and rectifying faults in cloud resources (infrastructure and applications) has become a critical issue that calls for proper attention and immediate solution. This is because cloud architecture is combination of a vast number of heterogeneous and geographically dispersed cloud modules connected via the Internet. Therefore,

the reliability of cloud application does not only depend on the system but also on the node and the availability of good Internet connection. Dealing with the occurrence of faults is more challenging in cloud computing than conventional client-server systems because of the dynamism of services and resources which can be automatically selected at run-time based on requests.

Table 2.3: Some Definitions of Cloud Computing

Definition	**Reference**
A style of computing where massively scalable IT-related capabilities are provided as a service across the Internet to multiple external customers.	(Gartner, 2017)
A pool of abstracted, highly scalable, and managed infrastructure capable of hosting end-customer applications and billed by consumption.	(Staten, 2008)
Cloud computing embraces cyber-infrastructure, and builds on virtualization, distributed computing, grid computing, utility computing, networking, and Web and software services.	(Armbrust, Fox, Griffith, Joseph, & Ranfy, 2009)
A type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.	(Buyya, Yeo, & Venugopal, 2008)
A large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the infrastructure provider by means of customized Service Level Agreements (SLAs).	(Vaquero, Rodero–Merino, Caceres, & Lindner, 2009)
A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (for instance networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.	(Mell & Grance, 2011)

Cloud computing, in recent years, has been able to transform a large part of the information technology (IT) industry, making software more attractive as a service and shaping the way hardware resources are designed and purchased. Developers with innovative ideas for new Internet services no longer require the large capital outlays in hardware to deploy their service or the human expense to operate it. They need not be concerned about overprovisioning for a service whose popularity does not meet their predictions, thus wasting costly resources, or under provisioning for one that becomes wildly popular, thus missing potential customers and revenue.

Cloud computing plays a significant role in providing computing services such as storage facilities, scalability, and more to businesses of varying sizes. It involves the collection of large pool of systems connected in private, public or hybrid networks, to provide dynamically scalable infrastructure for application, data and file storage (Nakkeeran, 2015). In cloud computing environment the clients outsource their data to the cloud that performs the computing operations and store the results (Bawa, 2012). This differs from traditional computing where users use portable storage media and rely on how robust or fast the machine is. Rather cloud computing offers a distributed environment that hosts hundreds of thousands of servers consisting of multiple memory modules, network cards, storage disks, and more.

2.5.1 Types of cloud service models

There are three main cloud service models according to Mell & Grance, (2011), and they are listed as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) as shown in Figure 2.5. Various existing deployment models were also discussed and they were listed as Private cloud, Public cloud, Community cloud and Hybrid cloud, as shown in Figure 2.6 and Figure 2.7

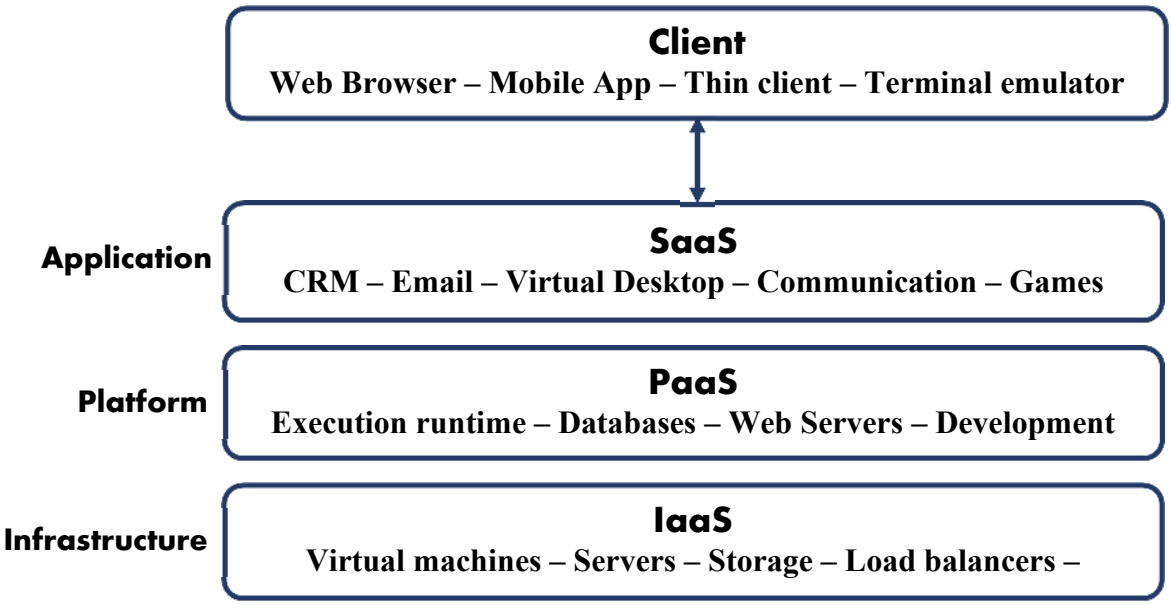

Figure 2.5: Cloud Service Models (Source: https://www.bluepiit.com)

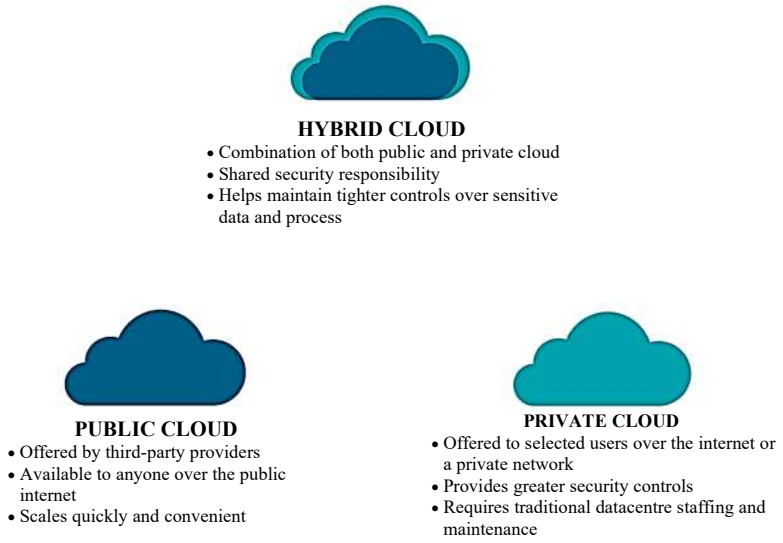

Figure 2.6: Public, private and hybrid cloud models:(Source: https://www.cloudflare.com/)

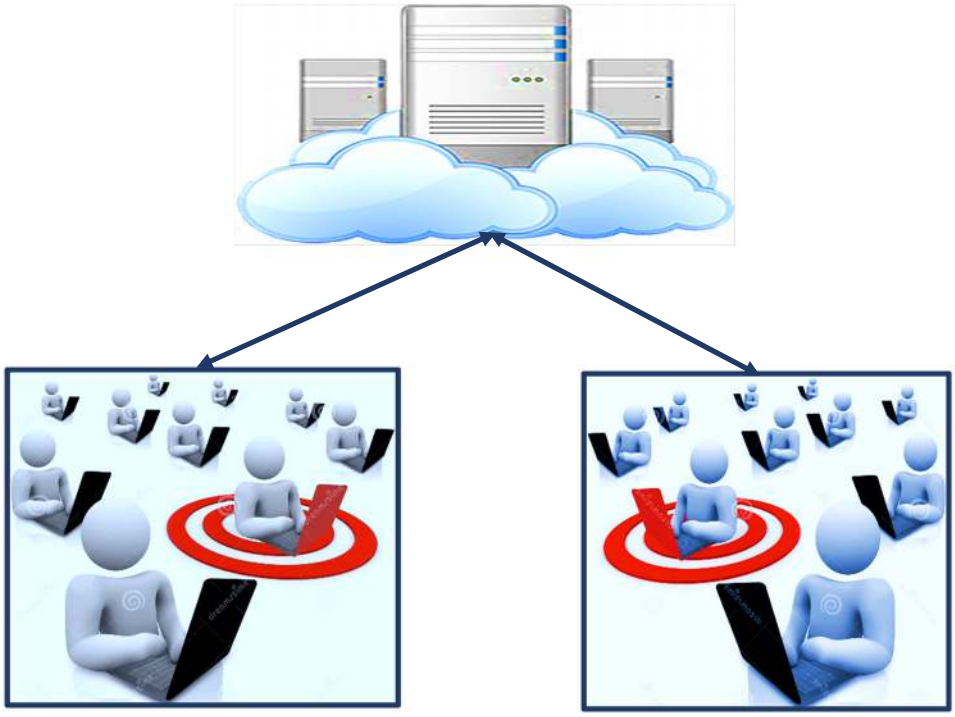

Figure 2.7: Community cloud model; Source: http://www.askurfriends.info/2013/06/cloud-deployment-models/

Mell and Grance, (2011), further explained these deployment models. The private cloud is provisioned strictly for usage by a single organisation that can have multiple users, and it may be owned, managed and controlled by the organisation. The community cloud is provisioned for use by a specific community of customers from organisation that have a common goal, and its ownership can be by the organisation in the community, a third party, or some combination of them, but in public cloud, the cloud infrastructure is provisioned for usage by the general public. Hybrid cloud is quite different as it is a composition of two or more cloud infrastructures, it may be a combination of private and community, private and public, or community and public, but are bound together by standardized technology that enables data and application portability.

Software as a Service (SaaS): The capability provided to the customer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (for instance, web-based email),

or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user- specific application configuration settings. The customer relationship management (CRM) solutions offered by SaaS have made both the global and the local enterprises alter their approach towards front office software solutions.

Platform as a Service (PaaS): The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

Infrastructure as a Service (IaaS): The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (for instance, host firewalls).

2.5.2 Advantages of SaaS to individuals and businesses

i. **Access to applications from anywhere.** Unlike on-premises software, which can be accessed only from a computer (or a network) it is installed on, SaaS solutions are cloud-based. Thus, can access them from anywhere there's internet access, be it a company's office or a hotel room.

ii. **Can be used from any device.** Cloud-based SaaS services can be accessed from any computer. Many SaaS solutions have mobile apps, so they can be accessed from mobile devices as well.

iii. **Automatic software updates:** Updating SaaS software is not necessary, as updates are carried out by the cloud service vendor. If there are any bugs or technical troubles, the vendor will fix them while the user focuses on his/her work instead of on software maintenance.

iv. **Low cost:** Compared to on-premises software, SaaS services are rather affordable. There's no need to pay for the whole IT infrastructure; user can pay only for the service at the scale needed. If a user needs extra functionalities, he/she can always update subscription.

v. **Simple adoption:** SaaS services are available out-of-the-box, so adopting them is quite simple. There's no need to install anything.

2.5.3 Advantages of PaaS to individuals and businesses

PaaS solutions are used mostly by software developers. PaaS provides an environment for developing, testing, and managing applications. Examples of PaaS services includes Heroku, Elastic Beanstalk (offered by Amazon Web Services), and Google App Engine.

PaaS provides a number of benefits to developers:

i. **Reduced development time:** PaaS services allow software developers to significantly reduce development time. Server-side components of the computing infrastructure (web servers, storage, networking resources, and so on.) are provided by a vendor, so development teams don't need to configure, maintain, or update them. Instead, developers can focus on delivering projects with top speed and quality.

ii. **Support for different programming languages:** PaaS cloud services usually support multiple programming languages, giving developers an opportunity to deliver various projects.

iii. **Easy collaboration for remote and distributed teams:** PaaS gives enormous collaboration capabilities to remote and distributed teams. Outsourcing and freelancing are common today, and many software development teams are comprised of specialists who live in different parts of the world. PaaS services allow them to access the same software architecture from anywhere and at any time.

iv. **High development capabilities without additional staff:** PaaS provides development companies with everything they need to create applications without the necessity of hiring additional staff. All hardware and middleware is provided, maintained, and upgraded by a PaaS vendor, which means businesses do not need staffs to configure servers and databases or deploy operating systems.

2.5.4 Advantages of IaaS to individuals and businesses

IaaS solutions can be used for multiple purposes. Unlike SaaS and PaaS, IaaS provides hardware infrastructure that can be used in a variety of ways. It is like having a set of tools that can be used for constructing the items needed.

Here are several scenarios where IaaS can be used:

i. **Website or application hosting:** Websites or applications can run with the help of IaaS; for example, using Elastic Compute Cloud from Amazon Web Services (Diginmotion, 2010).

ii. **Virtual Datacentres:** IaaS is the best solution for building virtual datacenters for large-scale enterprises that need an effective, scalable, and safe server environment.

iii. **Data analysis:** Analyzing huge amounts of data requires incredible computing power, and IaaS is the most economical way to get it. Companies use Infrastructure as a Service for data mining and analysis.

iv. **No expenses on hardware infrastructure:** IaaS vendors provide and maintain hardware infrastructure: servers, storage, and networking resources. This means that businesses don't need to invest in expensive hardware, which is a substantial cost savings as IT hardware infrastructure is rather pricey.

v. **Perfect scalability:** All cloud-based solutions are scalable; this is particularly true of Infrastructure as a Service, as additional resources are available to applications in case of higher demand.

vi. **Reliability and security.** Ensuring the safety of data is an IaaS vendor's responsibility. Hardware infrastructure is usually kept in specially designed datacentres, and a cloud provider guarantees security of your data.

A cloud service provider faces diverse cloud users with elastic demands. The cloud service provider's desires to meet the various demands in order to maximize profits corresponding to which diversified pricing policies are considered necessary (Subashini and Kavitha, 2011). Meanwhile, a variety of request durations creates a dynamic computing resource allocation problem(Du, 2012). Research in Vaquero *et al.*, (2009) discussed the cloud e-marketplace as the future of service delivery medium. Also, the work of Antonio *et al*., (2013) discussed cloud's configuration framework which considered that each market is created as a Business Line, implemented as a "swim lane" between the business line and the skin that interfaces directly with the user, as shown in Figure 2.8.

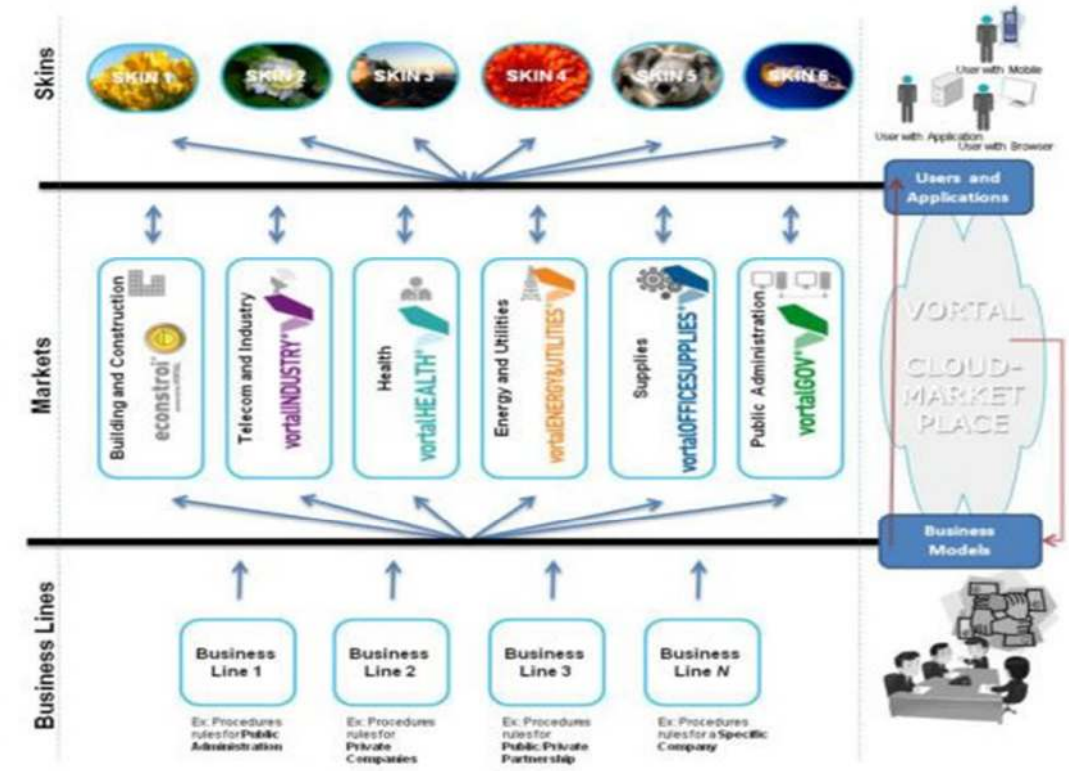

Figure 2.8: Cloud configuration framework; (Source: Antonio *et al*., 2013)

The work of Jennings and Stadler, (2014), highlighted compute resources, networking resources, storage resources and power resources as the main types of resources that a cloud vendor can provide for customers.

Stantchev, (2009), presented a general approach for evaluating non-functional QoS properties of individual cloud services. The approach is based on an architectural transparent “black box” methodology and comprises of the following steps: identifying benchmark, identifying configuration, running tests, analyzing results, and making recommendation. According to Duan, (2016), various performance metrics may be used for evaluating different features of cloud services, as shown in Table 2.4

Table 2.4: Performance metrics and description (Duan, 2016)

S/N	PERFORMANCE METRIC	DESCRIPTION
1	Response time (delay)	The latency time between service request and service completion
2	Throughput	The number of requests that can be processed by a service provider in a time limit
3	Service availability	The probability that a service request can be accepted by the service provider
4	System utilization	The percentage of system resources that are busy for service provisioning
5	System resilience	The stability of system performance over time especially under busty loads
6	System scalability	The ability of a system to perform well when its size is changed
7	System elasticity	The ability of a system to adapt to changes in its loads.

The benefits of cloud services as discussed in Gleb, (2017) are scalability, cost effectiveness, immediate availability and performance. According to Duan, (2016), developing effective methods for evaluating cloud service performance, and improving general levels of cloud computing are very important research problems that has attracted extensive attention from both academia and industry. The level architecture of cloud computing system is shown in Figure 2.9.

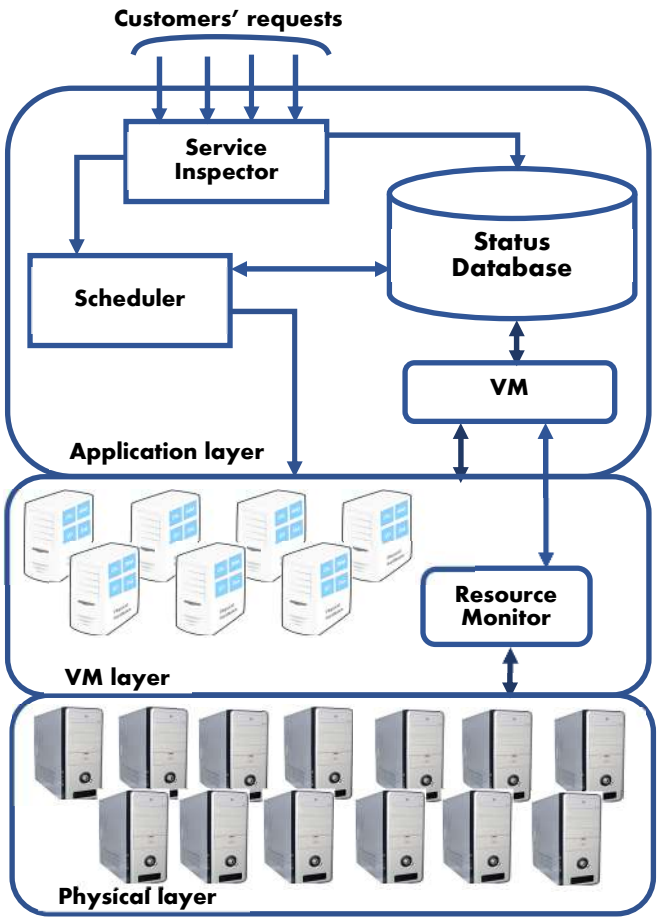

Figure 2.9: The level architecture of cloud computing system

Cost is one of the main concerns for cloud providers, mainly defined by the energy consumption of datacentres and cooling systems. Another major concern is the fulfillment of SLAs to ensure quality of service (Filho, *et al.,* 2018). The next few paragraphs present some works related to these concerns. Researchers have taken different types of approaches to tackling this challenging problem from various perspectives. Various approaches for effective cloud service delivery exist in the literature, for example; (Ezenwoke, *et al.*, (2017), Parikh, (2013), Sundareswaran *et al*., (2012) and Feng *et al*., (2019).

Parikh, (2013), concluded that an efficient resource allocation method should meet some criteria such as cost reduction, power reduction, energy reduction, QoS enhancement, and utilization of resources. The work of Ezenwoke *et al.*, (2017) considered the cloud e-marketplace as an ecosystem

that host heterogeneous cloud services from different cloud providers and supports collaboration. The author developed a cloud service selection framework that has a feature that aids numerous customers' requirements.

The work presented by Chaisiri, *et al.*, (2009) aimed to reduce costs for cloud computing customers by allowing them to trade-off between two different charging models, called reservation and on-demand plans. It proposed an optimal VM placement algorithm relying on multiple cloud providers to achieve the desired money saving for customers. A proposed decision-making process considers the uncertainty of demand in terms of number of required VMs and resource prices, assuming these random variables have a known distribution. It presents three resource usage phases: (i) resource reservation, where resources are not actually used, but only pre-booked; (ii) reserved resource utilization, where pre-booked resources start to be used and (iii) on-demand resource utilization, where resources beyond the reserved ones have to be allocated to meet a demand.

Cloud service providers are faced with several users with various demands. The cloud service provider desires to meet the various demands to maximize profits corresponding to which diversified pricing policies are considered necessary. Meanwhile, a variety of request durations creates a dynamic computing resource allocation problem (Du, 2012).

The work of Feng *et al*., (2019) focused on scheduling customer requests for SaaS providers with the explicit aim of cost minimization with dynamic demands handling by considering the customers' requests for an enterprise software services from a SaaS provider by agreeing to the pre-defined SLA clauses and submitting their QoS parameters. This research allowed customers to dynamically change their requirements and usage of cloud services. The cloud service provider's objective is to schedule a request such that its profit is maximized while the customers' QoS requirements are assured.

Three cost driven algorithms were proposed in this work to allow service providers to maximize profit: Base Algorithm: Maximizing the profit by minimizing the number of SLA violations, Propose Algorithm: Maximizing the profit by minimizing the cost by reusing VMs/Servers, which have maximum available space and also, maximizing the profit by minimizing the cost by reusing VMs/Servers, which have minimum available space.

A proposal for VM placement regarding QoS stated in the SLA and the impact of SLA violations on each cloud computing layer (IaaS, PaaS and SaaS) was presented in (Chaisiri *et al*., 2009). In this multi-domain (layer) cloud environment, an SLA violation on the IaaS, for instance, impacts all overlying layers. These violations may have a penalty, stated in the SLA, reducing service provider profit and customer satisfaction. The work focused on server consolidation, taking CPU usage as parameter for placement and migration decision making and proposing algorithms for VM selection and allocation. It aimed to reduce the number of VM migrations and accordingly avoid SLA violations. The authors tried to increase VMs availability and decrease Datacentre energy consumption by proposing a VM selection algorithm. The proposed VM selection algorithm chooses a VM to be migrated based on currently computed VM availability in decreasing order, selecting VMs with higher current availability first. The value is computed from the accumulated VM downtime, plus estimated downtime during migration. The proposed VM allocation algorithm chooses a destination host for selected VMs based on the host CPU usage and requirements of each VM.

The work of Feng *et al*., (2019), focused on profit maximization on Service provider's side by minimizing violations of pre-defined Service Level Agreements and re-using servers, the author measured the effectiveness of the algorithms based on some parameters: Arrival rate variation, Impact of QoS parameters, and others. However, the author only focused on QoS for customers and

did not consider optimizing the performance of the Servers to reduce customer's waiting time. Du, (2012) focused on allocating resources in a way that is sensible and cost effective by considering two cloud deployment models, the author called one deployment model the hybrid cloud which faces contract adopters who choose to rent exclusive Virtual Machine servers to handle the IT workload of regular hours but also buy public computing resources to handle peak hours' IT workload, and called another deployment model a public cloud which faces walk-in adopters who choose to only buy pooled resources when they need but willing to pay high prices.

Similarly, the author also assumed that the duration of each request for individual arrivals are independent and identically distributed non-negative random variables and are independent of the arrival process. The probability distributions of the pattern of arrivals are assumed to follow Poisson distributions with different arrival rates. The service distributions are assumed to follow exponential distributions. The public cloud follows "first come first served" in most of the case. However, when the hybrid adopters have heavy traffic that could not be handled by the exclusive Virtual Servers, those requests wait in the private Virtual Machine with a probability and also departs with a probability and jump into the line of public cloud with a priority. A delay cost per unit time per request is generated due to this congestion.

One effect of issues related with cloud fault tolerance is the increase in customers' waiting time. Prolonged waiting time has caused most service providers to lose customers. One noticeable thing is that as the number of customers increase, the waiting time also increases, and in addressing this waiting time challenge, some researchers have proposed the use of different methods; for example, the work of Ganga and Karthik, (2013), proposed the use of preemptive migration and software rejuvenation fault tolerant techniques which are used based on proactive fault tolerance policy. Park *et al*., (2010), used a Markov Chain model to collect the state of information as it changes

dynamically. Some researchers also proposed the use of additional servers as the method of solving the waiting time issue (Bari *et al.*, 2013).

While the issue of additional server has provided solution, however, the additional cost incurred by the cloud service provider is a great problem (Sheikholeslami and Nima, 2018). Therefore, balancing the trade-off between customers' waiting time and cloud service providers' cost is still a great challenge. This research proposes the checkpointing and replication techniques as the solution approach to these issues.

The replication method assumes that the likelihood of a single Virtual Machine failure is extremely higher than the occurrence of simultaneous failures of multiple VMs. It allows multiple VMs to start simultaneously executing redundant copies of the same request in order to preclude re-computation of it from scratch in a case of failure (Amoon, 2016). Therefore, the service can be efficiently provided to customers without affecting their QoS requirements in the presence of failures. In checkpointing, the cloud intermediately saves the execution state of both the currently executed request and the executing VM to a stable storage in order to minimize the recovery time in a case of failure (Zheng, *et al*., 2012). If a failure occurred, instead of restarting the request's execution from its early start, it will be started from the point in the computation where the last checkpoint was saved (Ganga & Karthik, 2013).