CS8603-DISTRIBUTED SYSTEMS
UNIT-I

PART-A

1 **Define distributed system.**
A distributed system is a collection of independent computers that appears to its users as a single coherent system. A distributed system is one in which components located at networked communicate and coordinate their actions only by passing message.

2 **List the characteristics of distributed system?**
- Programs are executed concurrently, support for resource sharing.
- Openness
- Concurrency
- Scalability
- Fault Tolerance (Reliability)
- Transparency
- Components can fail independently (isolation, crash)

3 **Mention the examples of distributed system.**
- The internet, intranet.
- Department computing cluster
- Corporate systems
- Cloud systems (e.g. Google, Microsoft, etc.)
- Mobile and ubiquitous computing

5 **Mention the challenges in distributed system.**
1. Heterogeneity
2. Openness
3. Security
4. Scalability
5. Failure handling
6. Concurrency
7.Transparency

6 **What are the Advantages of Distributed Systems?**
1. Performance 2.Distribution
3.Reliability (fault tolerance) 4.Incremental growth 5.Sharing of data/resources
6.Communication

7 **What are the Disadvantages of Distributed Systems?** MAY/JUNE 2016, **NOV/DEC 2016**
1.Difficulties of developing distributed software 2.Networking problems
3.Security problems

Software
Little software exists compared to PCs (for example) but the situation is improving with the cloud.

Networking
Still slow and can cause other problems (e.g., when disconnected)

Security
Data may be accessed by unauthorized users through network interfaces

Privacy
Data may be accessed securely but without the
owner's consent (significant issue in modern systems)

8 **What are the Applications of Distributed system?**

- Email
- News
- Multimedia information systems - video conferencing
- Airline reservation system
- BanKing system
- File downloads (BitTorrent)
- Messaging

9 **Write the different trends in distributed systems?**

- The emergence of pervasive networking technology;
- The emergence of ubiquitous computing coupled with the desire to support user mobility in distributed systems;

10 **Advantages of Distributed Systems vs. Centralized**

- Economics
- Speed
- Geographic and Responsibility Distribution
- Reliability
- Extendibility

11 **Write the Resource Sharing of Distributed system? NOV/DEC 2017**
1.Share hardware, 2.software, 3.data and information

Hardware Devices
Printers, disks, memory, sensors

Software Sharing
Compilers, libraries, toolkits, computational
Kernels

Data
Databases, files

12 **What are the Design issues of Distributed system?**

- Openness
- Resource Sharing
- Concurrency
- Scalability
- Fault-Tolerance
- Transparency
- High-Performance

13 **Write the issues arising from Distributed Systems?**

- Naming - How to uniquely identify resources.
- Communication - How to exchange data and information reliably with good performance.
- Software Structure - How to make software open, extensible, scalable, with high-performance.

- Workload Allocation - Where to perform computations and various services.
- Consistency Maintenance - How to Keep consistency at a reasonable cost.

14 **What is Communication in Distributed Systems?** Communication is an essential part of distributed systems - e.g., clients and servers must communicate for request and response. Communication normally involved - transfer of data from sender to receiver - synchronization among processes.

15 **What are types of Communication in Distributed Systems**

- Client-Server
- Group Multicast
- Function Shipping
- Performance of distributed systems depends critically on communication performance

16 **Distributed System Software Structure**

- It must be easy to add new services (flexibility, extensibility, openness requirements)
- Kernel is normally restricted to
- memory allocation
- process creation and scheduling
- interposes communication

17 **List any two resources of hardware and software, which can be shared in distributed systems with example. (NOV 2017)**

Hardware – Printer, Disks, Fax machine, Router, Modem.
Software – Application Programs, Shared Files, Shared Databases, Documents, Services.

18 **State the objectives of resurce sharing model APRIL/MAY 2018**

- resources are made available
- resources can be used
- service provider and user interact with each other
- accessing remote resources
- sharing them in a controlled and efficient way

19 **Write down the Principles of distributed systems APRIL/MAY 2018**

The principles of distributed computing, emphasizing the fundamental issues underlying the design of distributed systems and networks: communication, coordination, faulttolerance, locality, parallelism, self-organization,
synchronization, uncertainty

20 **What is clock skew and clock drift? APRIL/MAY 2018**

The instantaneous difference between the readings of any two clocks is called their skew. Clock drift means that they count time at different rates, and so diverge

21 **What is clocks drift rate?**

A clock's *driftrate* is the change in the offset (difference in reading) between the clock and a nominal
perfect reference clock per unit of time measured by the reference clock.

22 **What are the two modes of synchronization? Write their format?**

The two modes are:

External synchronization:

In order to Know at what time of day events occur at the processes in our distributed system – for example, for accountancy purposes – it is necessary to synchronize the processes' clocks, *Ci* , with an authoritative, external source of time. This is

externalsynchronization
For a synchronization bound D>0, and for a source S of UTC time,|S(t) –Ci(t)|<T, for i=1,2,…N and for all real times t in I. **Internal synchronization:**
If the clocks *Ci* are synchronized with one another to a Known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not
necessarily synchronized to an external source of time. This is *internal synchronization*. For a synchronization bound D>0,|Ci(t)-Cj(t)|<D,
for i,j=1,2,…N. and for all real times t in I.

23 **Explain Faultry and Crash Failure.**

A clock that does not Keep to whatever correctness conditions apply is defined to be *faulty*.
A clock's *crash failure* is said to occur when the clock stops ticKing altogether;any other clock failure is an *arbitrary failure*. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broKe the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes very large

24 **How the clock synchronization done in**
Cristian's method?

A single time server might fail, so they suggest the use of a group of synchronized servers It does not deal with faulty servers

25 **Explain Logical time and logical clocks. MAY/JUNE 2016**

Logical time

Lamport proposed a model of *logical time* that can be used to provide an ordering among the events at processes running in different computers in a distributed system.Logical time allows the order in which the messages are presented to be inferred without recourse to clocks.
Logical clocks • Lamport invented a simple mechanism by which the happenedbefore ordering can be captured numerically, called a *logical clock*. A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process *pi* Keeps its own logical clock, *Li* , which it uses to apply so- called *Lamport timestamps* to events. We denote the timestamp of event *e* at *pi* by *Li(e)* , and by *L(e*) we denote the timestamp of event *e* at
whatever process it occurred at.

26 **What is vector clock? Explain.**

Vector clocks • Mattern and Fidge developed vector clocks to overcome the shortcoming of Lamport's clocks: the fact that from *L*(*e)<L(e')*we cannot conclude that *e ->e*'
A vector clock for a system of *N* processes is an array of *N*integers. Each process Keeps its own vector clock, *Vi* , which it uses to timestamp local events. Like Lamport timestamps, processes piggybacK vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:
TaKing the componentwise maximum of two vector timestamps in this way is Known as a *merge* operation.

27 **Explain global states and consistent cuts with example.**

Global state of a distributed system consists of
–Local stateof each process: messages sent and messages received

–State of each channel:messages sent but not
received

28 **State the issues in Clocks. NOV/DEC 2018**

The Importance of Accurate Time on Computer Networks. The synchronization of time on computers and networks is often vitally important. Without it, the time on individual computers will slowly drift away from each other
at varying degrees until potentially each has a significantly different time

PART B

1 **Define distributed systems. What are the significant issues and challenges of the distributed systems? NOV/DEC 2017, APRIL/MAY 2018**

Designing a distributed system does not come as easy and straight forward. A number of challenges need to be overcome in order to get the ideal system. The major challenges in distributed systems are listed below:

1. Heterogeneity:

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- Hardware devices: computers, tablets, mobile phones, embedded devices, etc.
- Operating System: Ms Windows, Linux, Mac, Unix, etc.
- Network: Local network, the Internet, wireless network, satellite links, etc.
- Programming languages: Java, C/C++, Python, PHP, etc.
- Different roles of software developers, designers, system managers

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the
representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware : The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the differences in operating systems
and hardware

Heterogeneity and mobile code : The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

2. Transparency:

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. In other words, distributed systems designers must hide the complexity of the systems as much as they can. Some terms of

transparency in distributed systems are:
Access Hide differences in data representation and how a resource is accessed
Location Hide where a resource is located
Migration Hide that a resource may move to another location
Relocation Hide that a resource may be moved to another location while in use
Replication Hide that a resource may be copied in several places
Concurrency Hide that a resource may be shared by several competitive users
Failure Hide the failure and recovery of a resource
Persistence Hide whether a (software) resource is in memory or a disk

3. Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and reimplemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs. If the well-defined interfaces for a system are published, it is easier for developers to add new features or replace sub-systems in the future. Example: Twitter and Facebook have API that allows developers to develop theirs own software interactively.

4. Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

5. Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components:
confidentiality (protection against disclosure to unauthorized individuals)
integrity (protection against alteration or corruption),
availability for the authorized (protection against interference with the means to access the resources).

6. Scalability

Distributed systems must be scalable as the number of user increases. The scalability is defined by B. Clifford Neuman as A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity
Scalability has 3 dimensions:

- Size
  - Number of users and resources to be processed. Problem associated is overloading
- Geography
  - Distance between users and resources. Problem associated is communication reliability
- Administration

- As the size of distributed systems increases, many of the system needs to be controlled. Problem associated is administrative mess

7. Failure Handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. The handling of failures is particularly difficult.

2 **Enlighten the examples of distributed systems**.**MAY/JUNE 2016**

Examples of Distributed Systems

The goal of this section is to provide motivational examples of contemporary distributed systems and the great diversity of the associated applications.

As mentioned in the introduction, networks are everywhere and underpin many everyday services that we now take for granted: the Internet and he associated World Wide Web, web search, online gaming, email, social networks, eCommerce, etc. To illustrate this point further, consider Figure1.1, which describes a selected range of key commercial or social application sectors highlighting some of the associated established or emerging uses of distributed systems technology.

As can be seen, distributed systems encompass many of the most significant technological developments of recent years and hence an understanding of the underlying technology is absolutely central to a knowledge of modern computing. The figure also provides an initial insight into the wide range of applications in use today, from relatively localized systems (as found, for example, in a car or aircraft) toglobalscale systems involving millions of nodes, from data-centric services to processor-intensive tasks, from systems built from very small and relatively primitive sensors to those incorporating powerful computational elements, from embedded systems to ones that support a sophisticated interactive user experience, and so on. We now look at more specific examples of distributed systems to further illustrate the diversity and indeed complexity of distributed systems provision today.

Web search

Web search has emerged as a major growth industry in the last decade, with recent figures indicating that the global number of searches has risen to over 10 billion per calendar month. The task of a web search engine is to index the entire contents of the World Wide Web, encompassing a wide range of information styles including web pages, multimedia sources and (scanned) books. This is a very complex task, as current estimates state that the Web consists of over 63 billion pages and one trillion unique web

Finance and commerce - The growth of eCommerce as exemplified by companies such as Amazon and eBay, and underlying payments technologies such as PayPal; the associated emergence of online banking and trading and also complex information dissemination systems for financial markets.

The information society - The growth of the World Wide Web as a repository of information and knowledge; the development of web search engines such as Google and Yahoo to search this vast repository; the emergence of digital libraries and the large-scale digitization of legacy information sources such as books (for example, Google Books); the increasing significance of user-generated content through sites such as YouTube, Wikipedia and Flickr; the emergence of social networking through services such as Facebook and MySpace.

Creative industries and entertainment - The emergence of online gaming as a novel and highly interactive form of entertainment; the availability of music and film in the home through networked media centres and more widely in the Internet via downloadable or streaming

content; the role of user-generated content (as mentioned above) as a new form of creativity, for example via services such as YouTube; the creation of new forms of art and entertainment enabled by emergent (including networked) technologies.

Healthcare - The growth of health informatics as a discipline with its emphasis on online electronic patient records and related issues of privacy; the increasing role of telemedicine in supporting remote diagnosis or more advanced services such as remote surgery (including collaborative working between healthcare teams); the increasing application of networking and embedded systems technology in assisted living, for example for monitoring the elderly in their own homes.

Education - The emergence of e-learning through for example web-based tools such as virtual learning environments; associated support for distance learning; support for collaborative or community-based learning.

Transport and logistics - The use of location technologies such as GPS in route finding systems and more general traffic management systems; the modern car itself as an example of a complex distributed system (also applies to other forms of transport such as aircraft); the development of web-based map services such as MapQuest, Google Maps and Google Earth.

3 **Discuss the different trends in distributed systems**.

MAY/JUNE 2016, NOV/DEC 2016, NOV/DEC 2017, APRIL MAY 2018,

Trends in distributed systems

1. Trends in distributed systems • Significant changes in current distributed systems: – The emergence of pervasive technology – The emergence of ubiquitous & mobile computing – The increasing demand of multimedia technology – The view of distributed systems as a utility

2. Trends in distributed systems • Pervasive technology – Modern Internet – Collection of internetworked devices- wired & wireless – Pervasive resources and devices can be connected at any time and in any place

3. Trends in distributed systems intranet ISP desktop computer: backbone satellite link server: ☎ network link: ☎ ☎ ☎ A typical portion of the Internet

4. Trends in distributed systems • Mobile & ubiquitous computing – Small and portable devices are possible to be used within distributed systems • E.g. laptop computers, handheld devices, wearable devices, devices embedded in appliances – Mobile computing: portability of the devices and the ability to connect to networks in different places – Ubiquitous computing: small computing devices that available everywhere and are easily attached to networks

5. Trends in distributed systems Portable & handheld devices in a distributed system

6. Trends in distributed systems • Distributed multimedia systems – The use of multimedia contents in distributed systems • Multimedia support – Major benefits of multimedia support • Distributed multimedia computing can be accessed through desktop or mobile devices. E.g. live tv broadcast, video-on-demand, IP telephony, webcasting, etc.

7. Trends in distributed systems • Distributed computing as a utility – distributed resources as commodity or utility in similar as water and power. – Physical and logical service resources are rented rather than owned by the end users. • Physical resources: e.g. : storage and processing • Logical services: e.g. email, calendars – Cloud computing: distributed computing utility. A cloud is a set of internet-based application,

storage and computing services sufficient to support most users' needs
8. Trends in distributed systems Cloud computing
9. Trends in distributed systems • Cloud are implemented on cluster computers to provide the appropriate scale and performance required by such services – A cluster computer: a set of interconnected computers that cooperate closely to provide a single integrated high-performance computing capability – A blade server: a computer server that has been designed to minimize the use of physical space and energy
10. Trends in distributed systems • Grid Computing – Is a form of cloud computing – Authorized users share processing power, memory and data storage – Use to support scientific applications

8 **What are the different ways of synchronizing physical clocks? Explain**

Physical clock synchronization algorithm

Every computer contains a clock which is an electronic device that counts the oscillations in a crystal at a particular frequency. Synchronization of these physical clocks to some known high degree of accuracy is needed. This helps to measure the time relative to each local clock to determine order between events.

Physical clock synchronization algorithms can be classified as centralized and distributed.

1.Centralized clock synchronization algorithms

These have one node with a real-time receiver and are called time server node. The clock time of this node is regarded as correct and used as reference time.

The goal of this algorithm is to keep the clocks of all other nodes synchronized with time server node.

i. Cristian's Algorithm

- In this method each node periodically sends a message to the server. When the time server receives the message it responds with a message T, where T is the current time of server node.
- Assume the clock time of client be To when it sends the message and T1 when it receives the message from server. To and T1 are measured using same clock so best estimate of time for propagation is (T1-To)/2.
- When the reply is received at clients node, its clock is readjusted to T+(T1-T0)/2. There can be unpredictable variation in the message propagation time between the nodes hence (T1-T0)/2 is not good to be added to T for calculating current time.
- For this several measurements of T1-To are made and if these measurements exceed some threshold value then they are unreliable and discarded. The average of the remaining measurements is calculated and the minimum value is considered accurate and half of the calculated value is added to T.
- Advantage-It assumes that no additional information is available.
- Disadvantage- It restricts the number of measurements for estimating the value.

ii.The Berkley Algorithm

- This is an active time server approach where the time server periodically broadcasts its clock time and the other nodes receive the message to correct their own clocks.
- In this algorithm the time server periodically sends a message to all the computers in the group of computers. When this message is received each computer sends back its own clock value to the time server. The time server has a prior knowledge of the approximate time required for propagation of a message which is used to readjust the clock values. It then takes a fault tolerant average of clock values of all the computers.

The calculated average is the current time to which all clocks should be readjusted.
• The time server readjusts its own clock to this value and instead of sending the current time to other computers it sends the amount of time each computer needs for readjustment. This can be positive or negative value and is calculated based on the knowledge the time server has about the propagation of message.

2.Distributed algorithms

Distributed algorithms overcome the problems of centralized by internally synchronizing for better accuracy. One of the two approaches can be used:

i.Global Averaging Distributed Algorithms

• In this approach the clock process at each node broadcasts its local clock time in the form of a "resync" message at the beginning of every fixed-length resynchronization interval. This is done when its local time equals To+iR for some integer i, where To is a fixed time agreed by all nodes and R is a system parameter that depends on total nodes in a system.
• After broadcasting the clock value, the clock process of a node waits for time T which is determined by the algorithm.
• During this waiting the clock process collects the resync messages and the clock process records the time when the message is received which estimates the skew after the waiting is done. It then computes a fault-tolerant average of the estimated skew and uses it to correct the clocks.

ii.Localized Averaging Distributes Algorithms

• The global averaging algorithms do not scale as they need a network to support broadcast facility and a lot of message traffic is generated.
• Localized averaging algorithms overcome these drawbacks as the nodes in distributed systems are logically arranged in a pattern or ring.
• Each node exchanges its clock time with its neighbors and then sets its clock time to the average of its own clock time and of its neighbors.

9 **Explain Christian's method for synchronizing Clocks**

Cristian's Algorithm

Cristian's Algorithm is a clock synchronization algorithm is used to synchronize time with a time server by client processes. This algorithm works well with low-latency networks where Round Trip Time is short as compared to accuracy while redundancy prone distributed systems/applications do not go hand in hand with this algorithm. Here Round Trip Time refers to the time duration between start of a Request and end of corresponding Response.

Below is an illustration imitating working of cristian's algorithm:

Algorithm:

1) The process on the client machine sends the request for fetching clock time(time at server) to the Clock Server at time.
2) The Clock Server listens to the request made by the client process and returns the response in form of clock server time.
3) The client process fetches the response from the Clock Server at time and calculates the synchronised client clock time

Python Codes below illustrate the working of Cristian's algorithm:

Code below is used to initiate a prototype of a clock server on local machine:

filter_none
brightness_4

```
# Python3 program imitating a clock server
import socket
import datetime
# function used to initiate the Clock Server
def initiateClockServer():
s = socket.socket()
print("Socket successfully created")
# Server port
port = 8000
s.bind(('', port))
# Start listening to requests
s.listen(5)
print("Socket is listening...")
# Clock Server Running forever
while True:
# Establish connection with client
connection, address = s.accept()
print('Server connected to', address)
# Respond the client with server clock time
connection.send(str(
datetime.datetime.now()).encode())
# Close the connection with the client process
connection.close()
# Driver function
if __name__ == '__main__':
# Trigger the Clock Server
initiateClockServer()
```

Output:

Socket successfully created
Socket is listening...

10 **Explain Logical time and logical clocks.**

Logical time and logical clocks

Instead of synchronizing clocks, event ordering can be used
If two events occurred at the same process *pi* ($i = 1, 2, \ldots N$) then theyoccurred in the order observed by *pi*, that is order $\rightarrow_i$
when a message, *m* is sent between two processes, *send(m)* happened before *receive(m)*
Lamport[1978] generalized these two relationships into the **happened-before relation:** $e \rightarrow_i e'$
HB1: if $e \rightarrow_i e'$ in process *p*i, then $e \rightarrow e'$
HB2: for any message *m*, send(*m*) → receive(*m*)
HB3: if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$
Lamport's logical clocks
Each process *pi* has a logical clock ***Li***
a monotonically increasing software counter
not related to a physical clock
Apply Lamport timestamps to events with happened-before relation
LC1: *Li* is incremented by 1 before each event at process *pi*

LC2:
when process *pi* sends message *m*, it piggybacks $t = Li$
when *pj* receives *(m,t),* it sets *L*j := *max*(*L*j, *t*) and applies LC1 before timestamping the event *receive* (*m)*
$e \rightarrow e'$ implies $L(e)<L(e')$, but $L(e)<L(e')$ does not imply $e \rightarrow e'$
Totally ordered logical clocks
Some pairs of distinct events, generated by different processes, may have numerically identical Lamport timestamps
Different processes may have same Lamport time
Totally ordered logical clocks
If e is an event occurring at pi with local timestamp Ti, and if e' is an event occurring at pj with local timestamp Tj
Define global logical timestamps for the events to be (*Ti, i*) and (*Tj, j*)
Define (*Ti, i*) < (*Tj, j*) iff
Ti < *Tj* or*Ti* = *Tj* and *i* < *j*
No general physical significance since process identifiers are arbitrary
Vector clocks
Shortcoming of Lamport clocks:
L(e) < *L(e')* doesn't imply $e \rightarrow e'$
Vector clock: an array of N integers for a system of N processes
Each process keeps its own vector clock *Vi* to timestamp local events
Piggyback vector timestamps on messages
Rules for updating vector clocks:
Vi[*i*]] is the number of events that *pi* has timestamped
Viji] ($j \neq i$) is the number of events at *pj* that *pi* has been affected by VC1: Initially, *Vi*[*j*] := 0 for *pi*, *j*=1.. *N* (*N* processes)
VC2: before *pi* timestamps an event, *Vi*[*i*] := *Vi*[*i*]+1 VC3: *pi* piggybacks *t* = *Vi* on every message it sends
VC4: when *pi* receives a timestamp *t*, it sets *Vi*[*j*] := max(*Vi*[*j*] , *t*[*j*]) for *j*=1..*N* (merge operation)
Compare vector timestamps
V=V' iff V[j] = V'[j] for j=1..N
V>=V' iff V[j] <= V'[j] for j=1..N
V<V' iff V<= V' ^ V!=V'
$a \rightarrow f$ since V(a) < V(f)
c || e since neither V(c) <= V(e) nor V(e) <= V(c)

11 **Explain global states and consistent cuts with example**

Time and Global States

Overview

There are two formal models of distributed systems: synchronous and asynchronous. Synchronous distributed systems have the following characteristics:

- the time to execute each step of a process has known lower and upper bounds;
- each message transmitted over a channel is received within a known bounded time;
- each process has a local clock whose drift rate from real time has a known bound.

Asynchronous distributed systems, in contrast, guarantee no bounds on process execution speeds, message transmission delays, or clock drift rates. Most distributed systems we discuss,

including the Internet, are asynchronous systems.
Generally, timing is a challenging an important issue in building distributed systems. Consider a couple of examples:

- Suppose we want to build a distributed system to track the battery usage of a bunch of laptop computers and we'd like to record the percentage of the battery each has remaining at exactly 2pm.
- Suppose we want to build a distributed, real time auction and we want to know which of two bidders submitted their bid first.
- Suppose we want to debug a distributed system and we want to know whether variable x_1 in process p_1 ever differs by more than 50 from variable x_2 in process p_2.

In the first example, we would really like to synchronize the clocks of all participating computers and take a measurement of absolute time. In the second and third examples, knowing the absolute time is not as crucial as knowing the order in which events occurred.

Clock Synchronization

Every computer has a physical clock that counts oscillations of a crystal. This hardware clock is used by the computer's software clock to track the current time. However, the hardware clock is subject to *drift* -- the clock's frequency varies and the time becomes inaccurate. As a result, any two clocks are likely to be slightly different at any given time. The difference between two clocks is called their *skew*.
There are several methods for synchronizing physical clocks. *External synchronization* means that all computers in the system are synchronized with an external source of time (e.g., a UTC signal). *Internal synchronization* means that all computers in the system are synchronized with one another, but the time is not necessarily accurate with respect to UTC.
In a synchronous system, synchronization is straightforward since upper and lower bounds on the transmission time for a message are known. One process sends a message to another process indicating its current time, t. The second process sets its clock to $t + (max+min)/2$ where max and min are the upper and lower bounds for the message transmission time respectively. This guarantees that the skew is at most $(max-min)/2$.
Cristian's method for synchronization in asynchronous systems is similar, but does not rely on a predetermined max and min transmission time. Instead, a process p_1 requests the current time from another process p_2 and measures the RTT (T_{round}) of the request/reply. When p_1 receives the time t from p_2 it sets its time to $t + T_{round}/2$.
The Berkeley algorithm, developed for collections of computers running Berkeley UNIX, is an internal synchronization mechanism that works by electing a master to coordinate the synchronization. The master polls the other computers (called slaves) for their times, computes an average, and tells each computer by how much it should adjust its clock.
The Network Time Protocol (NTP) is yet another method for synchronizing clocks that uses a hierarchical architecture where he top level of the hierarchy (stratum 1) are servers connected to a UTC time source.

Logical Time

Physical time cannot be perfectly synchronized. Logical time provides a mechanism to define the *causal order* in which events occur at different processes. The ordering is based on the following:

- Two events occurring at the same process happen in the order in which they are observed by the process.
- If a message is sent from one process to another, the sending of the message happened

before the receiving of the message.
- If e occurred before e' and e' occurred before e" then e occurred before e".

"Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation." (→)
In the figure, a → b and c → d . Also, b → c and d → f , which means that a → f . However, we cannot say that a → e or vice versa; we say that they are *concurrent* (*a* || *e*).
A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process p_i keeps its own logical clock, L_i, which it uses to apply so-called *Lamport timestamps* to events.
Lamport clocks work as follows:
- LC1: L_i is incremented before each event is issued at p_i.
- LC2:
  - When a process p_i sends a message *m*, it piggybacks on *m* the value $t = L_i$.
  - On receiving (*m, t*), a process p_j computes $L_j := max(L_j, t)$ and then applies LC1 before timestamping the event *receive(m)*.

An example is shown below:
If e → e ' then L(e) < L(e'), but the converse is not true. Vector clocks address this problem. "A vector clock for a system of N processes is an array of N integers." Vector clocks are updated as follows:
VC1: Initially, $V_i[j] = 0$ for i, j = 1, 2, ..., N
VC2: Just before p_i timestamps an event, it sets $V_i[i]:=V_i[i]+1$.
VC3: p_i includes the value $t = V_i$ in every message it sends.
VC4: When p_i receives a timestamp t in a message, it sets $V_i[j]:=max(V_i[j], t[j])$, for 1, 2, ...N.
Taking the componentwise maximum of two vector timestamps in this way is known as a merge operation.
An example is shown below:
Vector timestamps are compared as follows:
V=V' iff V[j] = V'[j] for j = 1, 2, ..., N
V <= V' iff V[j] <=V'[j] for j = 1, 2, ..., N
V < V' iff V <= V' and V != V'
If e → e ' then V(e) < V(e') and if V(e) < V(e') then e → e ' .

Global States

It is often desirable to determine whether a particular property is true of a distributed system as it executes. We'd like to use logical time to construct a global view of the system state and determine whether a particular property is true. A few examples are as follows:
- Distributed garbage collection: Are there references to an object anywhere in the system? References may exist at the local process, at another process, or in the communication channel.
- Distributed deadlock detection: Is there a cycle in the graph of the "waits for" relationship between processes?
- Distributed termination detection: Has a distributed algorithm terminated?
- Distributed debugging: Example: given two processes p_1 and p_2 with variables x_1 and x_2 respectively, can we determine whether the condition $|x_1-x_2| > \delta$ is ever true.

In general, this problem is referred to as *Global Predicate Evaluation*. "A global state predicate is a function that maps from the set of global state of processes in the system ρ to {True,

False}."
- Safety - a predicate always evaluates to false. A given undesirable property (e.g., deadlock) never occurs.
- Liveness - a predicate eventually evaluates to true. A given desirable property (e.g., termination) eventually occurs.

Cuts

Because physical time cannot be perfectly synchronized in a distributed system it is not possible to gather the global state of the system at a particular time. Cuts provide the ability to "assemble a meaningful global state from local states recorded at different times".

Definitions:

- ρ is a system of N processes p_i (i = 1, 2, ..., N)
- history(p_i) = h_i = < e i 0 , e i 1 ,...>
- h i k =< e i 0 , e i 1 ,..., e i k > - a finite prefix of the process's history
- s i k is the state of the process p_i immediately before the kth event occurs
- All processes record sending and receiving of messages. If a process p_i records the sending of message m to process p_j and p_j has not recorded receipt of the message, then m is part of the state of the channel between p_i and p_j.
- A *global history* of ρ is the union of the individual process histories: H = h_0 ∪ h_1 ∪ h_2 ∪...∪h_{N-1}
- A *global state* can be formed by taking the set of states of the individual processes: S = (s_1, s_2, ..., s_N)
- A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories (see figure below).
- The *frontier* of the cut is the last state in each process.
- A cut is *consistent* if, for all events *e* and *e′*:
  - $(e \in C \text{ and } e' \rightarrow e) \Rightarrow e' \in C$
- A *consistent global state* is one that corresponds to a consistent cut.

Distributed Debugging

To further examine how you might produce consistent cuts, we'll use the distributed debugging example. Recall that we have several processes, each with a variable x_i. "The safety condition required in this example is $|x_i - x_j| <= \delta$ (i, j = 1, 2, ..., N)."

The algorithm we'll discuss is a centralized algorithm that determines post hoc whether the safety condition was ever violated. The processes in the system, p_1, p_2, ..., p_N, send their states to a passive monitoring process, p_0. p_0 is not part of the system. Based on the states collected, p_0 can evaluate the safety condition.

Collecting the state: The processes send their initial state to a monitoring process and send updates whenever relevant state changes, in this case the variable x_i. In addition, the processes need only send the value of x_i and a vector timestamp. The monitoring process maintains a an ordered queue (by the vector timestamps) for each process where it stores the state messages. It can then create consistent global states which it uses to evaluate the safety condition.

Let S = (s_1, s_2, ..., S_N) be a global state drawn from the state messages that the monitor process has received. Let V(s_i) be the vector timestamp of the state s_i received from p_i. Then it can be shown that S is a consistent global state if and only if:

V(s_i)[i] >= V(s_j)[i] for i, j = 1, 2, ..., N

12 **Explain an algorithm using multicast and logical clocks for mutual exclusion.**

Mutual exclusion in distributed system

Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.

Mutual exclusion in single computer system Vs. distributed system:

In single computer system, memory and other resources are shared between different processes. The status of shared resources and the status of users is easily available in the shared memory so with the help of shared variable (For example: Semaphores) mutual exclusion problem can be easily solved.

In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.

A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

Requirements of Mutual exclusion Algorithm:

- **No Deadlock:**

Two or more site should not endlessly wait for any message that will never arrive.

- **No Starvation:**

Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section

- **Fairness:**

Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.

- **Fault Tolerance:**

In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Solution to distributed mutual exclusion:

As we know shared variables or a local kernel can not be used to implement mutual exclusion in distributed systems. Message passing is a way to implement mutual exclusion. Below are the three approaches based on message passing to implement mutual exclusion in distributed systems:

1. **Token Based Algorithm:**
   - A unique **token** is shared among all the sites.
   - If a site possesses the unique token, it is allowed to enter its critical section
   - This approach uses sequence number to order requests for the critical section.
   - Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
   - This approach insures Mutual exclusion as the token is unique
   - **Example:**
   - Suzuki-Kasami's Broadcast Algorithm
2. **Non-token based approach:**
   - A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.

- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme
- **Example:**
- Lamport's algorithm, Ricart–Agrawala algorithm

3. **Quorum based approach:**
- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a **quorum**.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion

13 **Write short notes on locks with suitable example.**

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

There are four types of lock protocols available:

1. Simplistic lock protocol

It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

2. Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.

3. Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.

• In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
• In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.
There are two phases of 2PL:
Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.
In the below example, if lock conversion is allowed then the following phase can happen:
1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.
Example:
The following way shows how unlocking and locking work with 2-PL.
Transaction T1:
• **Growing phase:** from step 1-3
• **Shrinking phase:** from step 5-7
• **Lock point:** at 3
Transaction T2:
• **Growing phase:** from step 2-6
• **Shrinking phase:** from step 8-9
• **Lock point:** at 6
4. Strict Two-phase locking (Strict-2PL)
• The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
• The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
• Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
• Strict-2PL protocol does not have shrinking phase of lock release.
It does not have cascading abort as 2PL does.

UNIT-II
PART-A

1 **What is meant by group communication?** Group communication is a multicast operation is more appropriate- this is an operation that sends a single message from one process to each of the members of a group of process, usually in such a way that the membership of the group is transparent
to the sender.
2 **Difference between synchronous and asynchronous communication?**
In **synchronous form of communication**, the sending and receiving processes synchronize at every message. In this case, both send and receive are blocking operations. Whenever a send is issued the sending process is blocked until the corresponding receive is issued. Whenever receive is issued, the process blocks until a message arrives.

In **asynchronous form of communication**, the use of the send operation is non-blocking in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer and the transmission of the message proceeds in parallel with the sending process. The receive operation can
have blocking and non-blocking variants.

3 **What are the forms of message ordering paradigms?**

- FIFO
- non-FIFO
- Casual order
- Synchronous order

4 **What are the characterisitics of group communication?**

- Fault tolerance based on replicated server
- Finding the discovery servers from spontaneous networks
- Better performance through replicated data
- Propagation of event notification

5. **What are the two phases in obtaining a global snapshot?**

- First locally recording the snapshot at every process
- Second distributing the resultant global snapshot to all the initiators

6. **What are the two optimization techniques are provided to the Chandy-Lamport algorithm?**

- The first optimization combines snap-shots concurrently initiated by multiple processes into a single snapshot.
- This optimization is linked with the second optimization,which deals with the efficient distribution of the global snapshot.

7. **How a FIFO execution is implemented?**

- To implement a FIFO logical channel over a non-FIFO channel,a separate numbering scheme is used to sequence the messages.
- The sender assigns a sequence number and appends connection_id to each message and then transimmted then the receiver arranges the incoming messages according to the sender's sequence numbers and accepts "next" message s per sequence.

8. **What is Guard?**

A Guard Gi is a Boolean expression.If a Guard Gi evaluates to true then CLi is said to be enabled otherwise it is diabled.

9. **List the criteria to be met by a casual ordering protocol.**

- Safety
- Liveliness

10. **Write the drawback of centralized algorithm.**

- Single point of failure
- Congestion

11. **List the application of Casual order protocol.**

- Updating replicated data,
- Allocating requests in a fair manner
- Synchronizing multimedia streams

12 **What is the purpose of Chandy and Lamport algorithm?**

• Chandy and Lamport proposed a snapshot algorithm for determining global states of distributed systems.
• This algorithm records a set of process and channels as a snapshot for the process set.The recorded global state is consistent.

PART-B

1 **Explain in detail about asynchronous execution with synchronous communication**

Synchronous vs. Asynchronous

Definition:

• Synchronous communication: The calling party requests a service, and waits for the service to complete. Only when it receives the result of the service it continues with its work. A timeout may be defined, so that if the service does not finish within the defined period the call is assumed to have failed and the caller continues.
• Asynchronous communication: The calling party initiates a service call, but does not wait for the result. The caller immediately continues with its work without caring for the result. If the caller is interested in the result there are mechanisms which we'll discuss in the next paragraphs.

Be aware that the distinction between synchronous and asynchronous is highly dependent on the viewpoint. Often asynchronous is used in the sense of "the user interface must stay responsive all the time". This interpretation often leads to the wrong conclusion: "…and therefore every communication must be asynchronous". A non-blocking GUI usually has nothing to do with the low-level communication contracts and can be achieved by different means, e.g. parallel processing of the interactive and communication tasks. The truth is that synchronous communication on a certain level of abstraction can be implemented with asynchronous interfaces on another level of abstraction and vice versa, if needed.

File based communication is often considered to be asynchronous. One party writes a file but does not care if the other party is active, fetches the file or is able to process it. However it is possible to implement another layer of functionality so that the second (reading) party gives feedback, e.g. by writing a short result file, so that the first (writing) party can wait and poll for the result of the file processing. This layer introduces a synchronous communication over file exchange.

Communication over a database often is implemented by one party writing execution orders into a special table and the other party reads this table periodically and processes new entries, marking them as "done" or "failed" after execution. So far this is an asynchronous communication pattern. As soon as the first party waits for the result of the execution, this second layer introduces a synchronous communication pattern again.

The following chapters explain synchronous and asynchronous communication patterns in more detail, using web services as an example. The scenarios can also be used for other connectivity types.

Synchronous services are easy to implement, since they keep the complexity of the communication low by providing immediate feedback. They avoid the need to keep the context of a call on the client and server side, including e.g. the caller's address or a message id, beyond the lifetime of the request.

Nevertheless some circumstances may require implementing asynchronous calls:

• Expected round-trip durations are beyond time limits of the connection infrastructure (e.g. some web proxies close TCP connections after 2 minutes idle

time)
- Connections with a lack of stability (e.g. a dial-in network connection is not available all the time).
- The caller is not interested in the result of the call or cannot wait for the result for some reason, e.g. it must free its resources.

In some cases the delivery of the asynchronous request can be assured by some other mechanism, e.g. message queuing, storing the request in a file system or creation of a T4x job.

The simplest asynchronous message exchange pattern is called fire-and-forget and means that a message is sent but no feedback is required (at least on that level of abstraction!). The only possible feedback can come from the communication layer in case of an error in processing or sending the request, but never from the processing of the server.

If feedback on the server-side processing is required using a fire-and-forget transmission, some higher level implementation must add the necessary logic and data to establish a kind of "session" to link the feedback to the request. There are two possible patterns to implement this: Either the client repeatedly asks for the result of the processing on the server (polling) or the server calls a service of the client to report the feedback after it has finished processing (callback).

Polling causes potentially high network loads and is therefore not recommended. Nevertheless it has the advantage that the service provider (server) does not need to know about its clients and that no client needs to provide a service by itself.

On the contrary for the callback pattern, the receiver of the request (server) must by some means know how to send the feedback message and must know how to address the correct client (this information can be passed in the request or be stored statically). To collect the feedback some active instance on the caller's side must listen to receive the feedback message (which in turn can be a fire-and-forget message). So the caller must become a service provider ("server") by itself. Usually the client continues with its work after the request was fired instead of waiting. So there can be some interaction between the client and the callback instance to notify the client or the user of the arrival of the feedback. This interaction happens entirely on the client and is usually not a communication issue (instead you can imagine sending notification emails, notifying the GUI, push a workflow task to the user's inbox or similar actions).

As previously stated the implementation of the message transfer may use synchronous or asynchronous transfers on a lower level. In the fire-and-forget example, the request might be transferred via TCP, which implicitly acknowledges each message. Even if the acknowledgment is being implemented, higher levels might not be interested in it. For the callback and polling scenarios, each message might be acknowledged, but from a high level perspective, there are only fire-and-forget messages.

Asynchronous behavior can be implemented for a T4x server by writing the message (input parameters) to the file system (where the T4x scheduler will poll for it) or by creating a job at the T4x job server. If the caller wants to be informed about the execution's result, the T4x server needs to store the caller's response address and some context information to be able to report the result back. This has to be done in the service implementation. T4x as consumer can handle this by providing a callback service. Another possibility is the caller periodically polling for the execution result, e.g. by looking for a result file in the file system or by asking the T4x job server for the result of a job identified

by the job id. This can easily be done by providing an additional service asking for the job result.

2 **How Casual order and Total Order is implemented in Synchronization**

Causal ordering

Causal ordering is a vital tool for thinking about distributed systems.

Messages sent between machines may arrive zero or more times at any point after they are sent

This is the sole reason that building distributed systems is hard.

For example, because of this property it is impossible for two computers communicating over a network to agree on the exact time. You can send me a message saying "it is now 10:00:00" but I don't know how long it took for that message to arrive. We can send messages back and forth all day but we will never know for sure that we are synchronised.

If we can't agree on the time then we can't always agree on what order things happen in. Suppose I say "my user logged on at 10:00:00" and you say "my user logged on at 10:00:01". Maybe mine was first or maybe my clock is just fast relative to yours. The only way to know for sure is if something connects those two events. For example, if my user logged on and then sent your user an email and if you received that email before your user logged on then we know for sure that mine was first.

This concept is called causal ordering and is written like this:

A -> B (event A is causally ordered before event B)

Let's define it a little more formally. We model the world as follows: We have a number of machines on which we observe a series of events. These events are either specific to one machine (eg user input) or are communications between machines. We define the causal ordering of these events by three rules:

If A and B happen on the same machine and A happens before B then A -> B

If I send you some message M and you receive it then (send M) -> (recv M)

If A -> B and B -> C then A -> C

We are used to thinking of ordering by time which is a total order - every pair of events can be placed in some order. In contrast, causal ordering is only a partial order - sometimes events happen with no possible causal relationship i.e. not (A -> B or B -> A).

This image shows a nice way to picture these relationships.

On a single machine causal ordering is exactly the same as time ordering (actually, on a multi-core machine the situation is more complicated, but let's forget about that for now).

Between machines causal ordering is conveyed by messages. Since sending messages is the only way for machines to affect each other this gives rise to a nice property:

If not(A -> B) then A cannot possibly have caused B

Since we don't have a single global time this is the only thing that allows us to reason about causality in a distributed system. This is really important so let's say it again:

Communication bounds causality

The lack of a total global order is not just an accidental property of computer systems, it is a fundamental property of the laws of physics. I claimed that understanding causal order makes many other concepts much simpler. Let's skim over some examples.

Vector Clocks

Lamport clocks and Vector clocks are data-structures which efficiently approximate the causal ordering and so can be used by programs to reason about causality.

If A -> B then LC_A < LC_B

If VC_A < VC_B then A -> B
Different types of vector clock trade-off compression vs accuracy by storing smaller or larger portions of the causal history of an event.

Consistency

When mutable state is distributed over multiple machines each machine can receive update events at different times and in different orders. If the final state is dependent on the order of updates then the system must choose a single serialisation of the events, imposing a global total order. A distributed system is consistent exactly when the outside world can never observe two different serialisations.

CAP Theorem

The CAP (Consistency-Availability-Partition) theorem also boils down to causality. When a machine in a distributed system is asked to perform an action that depends on its current state it must decide that state by choosing a serialisation of the events it has seen. It has two options:

- Choose a serialisation of its current events immediately
- Wait until it is sure it has seen all concurrent events before choosing a serialisation

The first choice risks violating consistency if some other machine makes the same choice with a different set of events. The second violates availability by waiting for every other machine that could possibly have received a conflicting event before performing the requested action. There is no need for an actual network partition to happen - the trade-off between availability and consistency exists whenever communication between components is not instant. We can state this even more simply:

Ordering requires waiting

Even your hardware cannot escape this law. It provides the illusion of synchronous access to memory at the cost of availabilty. If you want to write fast parallel programs then you need to understand the messaging model used by the underlying hardware.

Eventual Consistency

A system is eventually consistent if the final state of each machine is the same regardless of how we choose to serialise update events. An eventually consistent system allows us to sacrifice consistency for availability without having the state of different machines diverge irreparably. It doesn't save us from having the outside world see different serialisations of update events. It is also difficult to construct eventually consistent data structures and to reason about their composition.

3 **What is group communication? What are the Key areas of applications of group communication? Explain the programming model for group communication.APRIL/MAY 2018**

Group Communication

A group is an operating system abstraction for a collective of related processes. A set of cooperative processes may, for example, form a group to provide an extendable, efficient, available and reliable service. The group abstraction allows member processes to perform computation on different hosts while providing support for communication and synchronisation between them.

The term multicast means the use of a single communication primitive to send a message to a specific set of processes rather than using a collection of individual point to point message primitives. This is in contrast with the term broadcast which means the message is addressed to every host or process.

A consensus protocol allows a group of participating processes to reach a common decision, based on their initial inputs, despite failures.
A reliable multicast protocol allows a group of processes to agree on a set of messages received by the group. Each message should be received by all members of the group or by none. The order of these messages may be important for some applications. A reliable multicast protocol is not concerned with message ordering, only message delivery guarantees. Ordered delivery protocols can be implemented on top of a reliable multicast service.
Multicast algorithms can be built on top of lower-level communication primitives such as point-to-point sends and receives or perhaps by availing of specific network mechanisms designed for this purpose.
The management of a group needs an efficient and reliable multicast communication mechanism to allow clients obtain services from the group and ensure consistency among servers in the presence of failures. Consider the following two scenarios:-
A client wishes to obtain a service which can be performed by any member of the group without affecting the state of the service.
A client wishes to obtain a service which must be performed by each member of the group.
In the first case, the client can accept a response to its multicast from any member of the group as long as at least one responds. The communication system need only guarantee delivery of the multicast to a nonfaulty process of the group on a best-effort basis. In the second case, the all-or-none atomic delivery requirements requires that the multicast needs to be buffered until it is committed and subsequently delivered to the application process, and so incurs additional latency.
Failure may occur during a multicast at the recipient processes, the communication links or the originating process.
Failures at the recipient processes and on the communication links can be detected by the originating process using standard time-out mechanisms or message acknowledgements. The multicast can be aborted by the originator, or the service group membership may be dynamically adjusted to exclude the failed processes and the multicast can be continued.
If the originator fails during the multicast, there are two possible outcomes. Either the message has not have arrived at any destination or it has arrived at some. In the first case, no process can be aware of the originator's intention and so the multicast must be aborted. In the second case it may be possible to complete the multicast by selecting one of the recipients as the new originator. The recipients would have to buffer messages until safe for delivery in case they were called on for this role.
A reliable multicast protocol imposes no restriction on the order in which messages are delivered to group processes. Given that multicasts may be in progress by a number of originators simultaneously, the messages may arrive at different processes in a group in different orders. Also, a single originator may have a number of simultaneous multicasts in progress or may have issued a sequence of multicast messages whose ordering we might like preserved at the recipients. Ideally, multicast messages should be delivered instantaneously in the real-time order they were sent, but this is unrealistic as there is no global time and message transmission has a possibly significant and variable latency.
A number of possible scenarios are given below which may require different levels of ordering semantics. G and s represent groups and message sources. s may be inside or

outside a group. Note that group membership may overlap with other groups, that is, processes may be members of more than one group.

Ordered Reliable Multicasts

A FIFO ordered protocol guarantees that messages by the same sender are delivered in the order that they were sent. That is, if a process multicasts a message m before it multicasts a message m', then no correct process receives m' unless it has previously received m. To implement this, messages can be assigned sequence numbers which define an ordering on messages from a single source. Some applications may require the context of previously multicast messages from an originator before interpreting the originator's latest message correctly.

However, the content of message m may also depend on messages that the sender of m received from other sources before sending m. The application may require that the context which could have caused or affected the content of m be delivered at all destinations of m, before m. For example, in a network news application, user A broadcasts an article. User B at a different site receives the article and broadcasts a response. User C can only interpret the response if the original article is delivered first at their site. Two messages are said to be causally related if one message is generated after receipt of the other. Causal order is a strengthening of FIFO ordering which ensures that a message is not delivered until all messages it depends on have been delivered.

This causal dependence relation is more formally specified as follows:- An execution of a multicast or receive primitive by a process is called an event.

Event e causally precedes event f (i.e. happened before), ($e \rightarrow f$), if an only if:

1. A process executes both e and f in that order, or
2. e is the multicast of message m and f is the receipt of m, or
3. there is an event h, such that $e \rightarrow h$ and $h \rightarrow f$.

A causal protocol then guarantees that if the broadcast of message m causally precedes the broadcast of m', then no correct process receives m' unless it has previously received m. The definition of causal ordering does not determine the delivery order of messages which are not causally related. Consider a replicated database application with two copies of a bank account x residing at different sites. A client side process at one site sends a multicast to the database to lodge £100 to account x. At another site simultaneously, a client side process initiates a multicast to add 10% interest to the current balance of x. For consistency, all database replicas should apply the two updates in the same sequence. As these two messages are not causally related, a causal broadcast would allow the update messages to x to be delivered in different sequences at the replicas.

Total Ordering guarantees that all correct processes receive all messages in the same order. That is, if correct processes p and q both receive messages m and m', then p receives m before m' if and only if q receives m before m'. The multicast is atomic across all members of the group.

Note that this definition of a totally ordered broadcast does not require that messages be delivered in Causal Order or even FIFO Order, so it is not stronger than these orderings. For example, if a process suffers a transient failure during the broadcast of message m, and subsequently broadcasts m', a totally ordered protocol will guarantee only that processes receive m'.

FIFO or Causal Ordering can be combined with Total Ordering if required.

All reliable multicasts have the following three properties.

Validity: If a correct process multicasts a message m, then all correct processes eventually receive m.
Agreement: If a correct process receives a message m, then all correct processes eventually receive m.
Integrity: For any message m, every correct process receives m at most once and only if m was multicast by the sender of m.
The protocols only differ in the strength of their message delivery order requirements.
Multicast Algorithms
In the algorithms to follow, R stands for Reliable Multicast, F for FIFO, C for Causal and A for Atomic.
In an asynchronous system where a reliable link exists between every pair of processes, the algorithm below demonstrates how a Reliable multicast can be achieved.
Every process p executes the following:-
multicast(R, m):
tag m with sender(m) and seq#(m)
send(m) to all group including p
The receive(R, m) occurs as follows:
upon arrival(m) do
if p has not previously executed receive(R, m) then
if sender(m) <> p then
send(m) to all group
receive(R, m)
It is easy to use Reliable Multicast to build a FIFO Multicast algorithm. To F-multicast a message m, a process q simply R-multicasts m. The difference is at the receiver which orders the delivery.
multicast(F, m):
multicast(R, m)
If m is the ith message from q, then m is tagged sender(m)=q and seq#(m)=i.
For each q, every process p maintains a counter next[q] that indicates the sequence number of the next F-multicast from q that p is willing to F-deliver. Incoming messages are placed in a message bag from which messages that can be FIFO delivered (according to the value of next[q]) are removed.
Every process p initialises:-
msgbag = ∅
next[q] = 1 for all q
The receive(R, m) occurs as follows:
upon arrival(R, m) do
q := sender(m);
msgbag := msgbag ∪ {m};
while (∃ m'∈ msgbag: sender(m')=q and seq#(m')=next[q]) do
receive(F, m')
next[q] = next[q]+1;
msgbag := msgbag - {m'}
A Causal multicast algorithm can be constructed on top of a FIFO multicast.
To C-multicast a message m, a process p uses the FIFO multicast algorithm to F-multicast the sequence of messages that p has causally received since its previous C-broadcast

followed by the message m. A process q receives all of the messages in the sequence only which it previously has not received.
Initialisation:
prevReceives = ⊥
multicast(C, m):
multicast(F, <prevReceives || m>) /* || is list concatenation operator */
prevReceives = ⊥
The receive(C, m) occurs as follows:
upon arrival(F, <m1,m2,…mn>) do
for i := 1 to n do
if p has not received mi then
receive(C, mi)
prevReceives := prevReceives || mi;
One of the fundamental results about fault-tolerant distributed computing is the impossibility of achieving consensus in asynchronous systems which suffer crash failures. This is primarily due to the fact that it is impossible to distinguish between a process that has crashed and one that is extremely slow. The consensus problem can be easily reduced to implementing atomic multicast. For example, to propose a value, a process A-multicasts it. To decide on a value, each process picks the value of the first message that is Areceived. By total order of atomic broadcasts, all processes agree on the same value.
Consensus and Atomic Multicast are therefore equivalent in asynchronous systems with crash failures and so there are no deterministic Atomic Multicast protocols (total ordering protocols) for such systems.
Atomic multicast can be deterministically implemented in synchronous systems where upper bounds on message latency exist.
If we assume no failures, the algorithm given below can be used to implement a totally ordered multicast.
Each site maintains a 'local clock'. A clock doesn't necessarily have to supply the exact time, but could be implemented simply by a counter which is incremented after each send or receive event that occurs at the site, so that successive events have different 'times'. The algorithm executes in two phases. During the first phase the originator multicasts the message to all destinations and awaits a reply from each. Each receiver queues the message and assigns it a proposed timestamp based on the value of its local clock. This timestamp is returned to the originator. The originator collects all the proposed timestamps for the multicast and selects the highest. During the second phase of the algorithm, the originator commits the multicast by sending the final chosen timestamp to all destinations. Each receiver then marks the message as deliverable in its queue. The message queue is ordered on the value of the timestamps associated with each message each time a timestamp is updated. When a message gets to the top of the queue and is deliverable it may be delivered immediately to the application.

UNIT-III

PART-A

1 **What is distributed deadlock? Explain with example.** With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock. Most deadlock detection schemes operate by finding cycles in the transaction wait- for graph. In a distributedsystem involving multiple servers being accessed by multiple transactions, a global wait-for graph can in theory be constructed from the local ones. There can be a cycle in the global wait-for graph that is not in any single local one – that is, there can be a *distributed deadlock*

2 **Explain the 'snapshot' algorithm of Lamport.**

The 'snapshot' algorithm of Chandy and Lamport describe a 'snapshot' algorithm for determining global states of distributed systems, which we now present. The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes *pi*($i = 1,2,.N$) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent

3 **Explain phantom deadlocks.**

A deadlock that is 'detected' but is not really a deadlock is called phantom deadlock. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from on server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. Ja this procedure will take some time, there is a chance that one of the transactions
that Holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

4 **Explain edge chasing deadlock detection technique in distributed systems.**

A distributed approach to deadlock detection uses a technique called *edge chasing* or *path pushing*. In this approach, the global wait-for graph is not constructed, but each of the servers involved has Knowledge about some of its edges.

The servers attempt to find cycles by forwarding messages called *probes*, which follow the edges of the graph throughout the distributed system. A probe message consists of transaction wait-for relationships representing a path in the global
wait-for graph.

5. **Define Distributed Mutual Exclusion.**

A condition in which there is a set of processes ,only one of which is able to access a given resource or perform a given function at a time.

6. **Compare Deadlock and Starvation**

- Deadlock happens when two or more process indefinitely gets stopped when it attempts to enter or exit the critical section
- Starvation is the indefinite postponement of entry for a process that has requested it.Without Deadlock Starvation may also occur. No starvation leads to fairness condition.

7. **What are the approaches to implement distributed mutual exclusion.**

- Token based approach
- Non Token based approach
- Quorum based approach

8. **What are the three states of Mutual Exclusion?**

It is of three states

1.Requesting Control Section

2.Executing Control Section
3.Or Neither requesting nor executing control section(idle)

9. **Define Throughput.**

The rate at which the system executes request for the critical section if synchronization delay is SD and E is the average critical section execution time,then the throughput is given by the equation

System Throughput=1/(SD+E)

10 **Define Response time.**

The time interval is the request wait for its control section execution to be over after its request message have been sent out .It does not include the time request waits at a site before its request message have been sent out.

PART-B

1 **Explain the 'snapshot' algorithm of Lamport**. **APRIL/MAY 2017, APRIL/MAY 2018**

Chandy–Lamport's global state recording algorithm

Each distributed system has a number of processes running on a number of different physical servers. These processes communicate with each other via communication channels using text messaging. These processes neither have a shared memory nor a common physical clock, this makes the process of determining the instantaneous global state difficult.

A process could record it own local state at a given time but the messages that are in transit (on its way to be delivered) would not be included in the recorded state and hence the actual state of the system would be incorrect after the time in transit message is delivered.

Chandy and **Lamport** were the first to propose a algorithm to capture consistent global state of a distributed system. The main idea behind proposed algorithm is that if we know that all message that hat have been sent by one process have been received by another then we can record the global state of the system.

Any process in the distributed system can initiate this global state recording algorithm using a special message called **MARKER**. This marker traverse the distributed system across all communication channel and cause each process to record its own state. In the end, the state of entire system (Global state) is recorded. This algorithm does not interfere with normal execution of processes.

Assumptions of the algorithm:

- There are finite number of processes in the distributed system and they do not share memory and clocks.
- There are finite number of communication channels and they are unidirectional and FIFO ordered.
- There exists a communication path between any two processes in the system
- On a channel, messages are received in the same order as they are sent.

Algorithm:

- **Marker sending rule for a process *P* :**
  - Process **p** records its own local state
  - For each outgoing channel **C** from process **P**, **P** sends marker along **C** before sending any other messages along **C**.

(**Note:** Process Q will receive this marker on his incoming channel C1.)

- **Marker receiving rule for a process *Q* :**

o If process **Q** has not yet recorded its own local state then

- Record the state of incoming channel C1 as an empty sequence or null.
- After recording the state of incoming channel C1, process **Q** Follows

the marker sending rule

o If process **Q** has already recorded its state

- Record the state of incoming channel **C1** as the sequence of messages

received along channel C1 after the state of **Q** was recorded and before **Q** received the marker along C1 from process P.

Need of taking snapshot or recording global state of the system:

- **Checkpointing:** It helps in creating checkpoint. If somehow application fails, this checkpoint can be used re
- **Garbage collection:** It can be used to remove objects that do not have any references.
- It can be used in deadlock and termination detection.
- It is also helpful in other debugging.

2 **Explain the bully algorithm**

Bully Algorithm

This algorithm has three main components given below.

- Coordinator – Announce about himself.
- Election – Announces the election.
- Reply – Acknowledge the request.

Let's say the scenario is, we have 6 process numbered as 1, 2, 3, 4, 5, 6 and also, the priority or process number are also in the same order, therefore the process 6 is the highest process number. The process are shown below; Circles are the processes and the square boxes are their numbers.

Now, suppose the case is Process 6 has crashed and other processes are active. The crashing has been noticed by process 2. It finds out that the Process 6 is longer responding to the request. In this case, Process 2 will start a fresh election.

Process 2 sends an election message to the process, which has the highest number. In our case, the processes are 3, 4, 5, 6. Now, as Process 6 is down or fails, it will definitely not respond to the election message.

Process 3, 4, 5 are active and therefore they respond with a reply or acknowledgement message to Process 2.

If no one will respond to the request message by Process 2, it will win the election. Now, the election will be initiated by the next highest number. In our case, it is Process 3, which will send the election message to Process 4, 5 and 6. As Process 6 is down, it will again not respond to the request message. Again, if no one will respond, then Process 3 will win the election.

It will be the same process for Process 4 and it will be the next initiator to conduct the election. Now, the chance came to Process 5, as it is the highest in all processes. Also, because Process 6 is down. In this case, Process 5 will win the election and will send this victory message to all.

Meanwhile, Process 6 came back from the down state to active state. It will definitely hold the election, as it is the highest among the processes in the system. It will win the election, which is based on the highest number and control over the Coordinator job.

Whenever the highest number process is recovered from the down state, it holds the election

and win the election. Also, it bullies the other process to get in submission.

The complexity of this algorithm is given below.

- Best Case - n-2
- Worst Case - 0(n^2)

3 **What is a deadlock? How deadlock can be recovered? Explain distributed dead locks.**

Deadlock Detection

1. If resources have single instance:

In this case for Deadlock detection we can run an algorithm to check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is the sufficient condition for deadlock.

In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. If there are multiple instances of resources:

Detection of the cycle is necessary but not sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

DeadlockRecovery

A traditional operating system such as Windows doesn't deal with deadlock recovery as it is time and space consuming process. Real-time operating systems use Deadlock recovery.

Recovery method

1. **Killing the process:** killing all the process involved in the deadlock. Killing process one by one. After killing each process check for deadlock again keep repeating the process till system recover from deadlock.
2. **Resource Preemption:** Resources are preempted from the processes involved in the deadlock, preempted resources are allocated to other processes so that there is a possibility of recovering the system from deadlock. In this case, the system goes into starvation.

UNIT-IV
PART-A

1 **Define Roll back recovery?**

Roll back recovery is defines as a system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure.

2 **What is a local checkpoint?**

A local checkpoint is a snapshot of the state of the process at a given instance and the event of recording the state of a process is called local check pointing.

3 **What are the types of messages in recovery?**

In-transit messages
Lost messages
Delayed messages
Orphan messages
Duplicate messages

4 **What is an Orphan message?**

Messages with receive recorded but message send not recorded are called orphan messages.

5 **Classify the checkpoint-based rollback recovery techniques.**

- Uncoordinated check pointing

- Coordinated check pointing
- Communication induced check pointing

6 **What is the necessity of Uncoordinated check pointing?**

Each process takes its checkpoints independently. This eliminates the synchronizat ion overhead.

7 **What are the types of communication-induced check pointing?**

- Model-based check pointing
- Index-based check pointing

8 **What are the advantages of pessimistic logging?**

- Immediate output commit
- Restart from most recent checkpoint
- Recovery limited to failed process
- Simple garbage collection

9 **State the role of Index-based checkpointing.**

Index based communication-induced check pointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes from a consistent state.

10 **What are the types of rollback-recovery protocols?**

- Pessimistic logging,
- Optimistic logging
- Causal logging protocols

11 **Compare agreement problem and the consensus problem**

The difference between the agreement problem and the consensus problem is that in the agreement problem, a single process has the initial value, whereas in the consensus problem,all processes have an initial value.

12 **Write about Reliable Broadcast (RTB)**

- RTB requires recognition of a failure,even if no message is sent.
- It is required to distinguish between a failed process and a slow process
- RTB requires eventual delivery of messages, even if sender fails before sending.

In this case, a null message needs to get sent. In RB, this condition is not there.

PART-B

1 **Explain about the checkpoint based recovery.**

Introduction

Checkpoint-Recovery is a common technique for imbuing a program or system with fault tolerant qualities, and grew from the ideas used in systems which employ transaction processing . It allows systems to recover after some fault interrupts the system, and causes the task to fail, or be aborted in some way. While many systems employ the technique to minimize lost processing time, it can be used more broadly to tolerate and recover from faults in a critical application or task.

The basic idea behind checkpoint-recover is the saving and restoration of system state. By saving the current state of the system periodically or before critical code sections, it provides the baseline information needed for the restoration of lost state in the event of a system failure. While the cost of checkpoint-recovery can be high, by using techniques like memory exclusion, and by designing a system to have as small a critical state as possible may minimize the cost of

checkpointing enough to be useful in even cost sensitive embedded applications.
When a system is checkpointed, the state of the entire system is saved to non-volatile storage. The checkpointing mechanism takes a snapshot of the system state and stores the data on some non-volatile storage medium. Clearly, the cost of a checkpoint will vary with the amount of state required to be saved and the bandwidth available to the storage mechanism being used to save the state.
In the event of a system failure, the internal state of the system can be restored, and it can continue service from the point at which its state was last saved. Typically this involves restarting the failed task or system, and providing some parameter indicating that there is state to be recovered. Depending on the task complexity, the amount of state, and the bandwidth to the storage device this process could take from a fraction of a second to many seconds.
This technique provides protection against the transient fault model. Typically upon state restoration the system will continue processing in an identical manner as it did previously. This will tolerate any transient fault, however if the fault was caused by a design error, then the system will continue to fail and recover endlessly. In some cases, this may be the most important type of fault to guard against, but not in every case.
Unfortunately, it has only limited utility in the presence of a software design fault. Consider for instance a system which performs control calculations, one of which is to divide a temperature reading into some value. Since the specification requires the instrument to read out in degrees Kelvin (absolute temperature), a temperature of 0 is not possible. In this case the programmer (realizing this) fails to check for zero prior to performing the divide. The system works well for a few months, but then the temperature gauge fails. The manufacturer realizes that a 0K temperature is not possible, and decides that the gauge should fail low, since a result of 0 is obviously indicative of a failure. The system faults, and attempts to recover its state. Unfortunately, it reaches the divide instruction and faults, and continues to recover and fault until some human intervention occurs. The point here is not that there should be redundant temperature sensors, but that the most common forms of checkpoint and recovery are not effective against some classes of failures.

KeyConcepts

The basic mechanism of checkpoint-recovery consists of three key ideas - the saving and restoration of executive state, and the detection of the need to restore system state. Additionally, for more complex distributed embedded systems, the checkpoint-recovery mechanism can be used to migrate processes off individual nodes

Saving executive state

A snapshot of the complete program state may be scheduled periodically during program execution. Typically this is accomplished by pausing the operation of the process whose state is to be saved, and copying the memory pages into non-volatile storage. While this can be accomplished by using freely available checkpoint-recovery libraries, it may be more efficient to build a customized mechanism into the system to be protected.
Between full snapshots, or even in place of all but the first complete shot, only that state which has changed may be saved. This is known as incremental checkpointing, and can be thought of in the same way as incremental backups of hard disks. The basic idea here is to minimize the cost of checkpointing, both in terms of the time required and the space (on non-volatile storage).
Not all program state may need to be saved. System designers may find it more efficient to build in mechanisms to regenerate state internally, based on a smaller set of saved state. Although this technique might be difficult for some applications, it has the benefit of having the potential to

save both time and space during both the checkpoint and recovery operations.
A technique known as memory exclusion allows a program to notify the checkpoint algorithm which memory areas are state critical and which are not. This technique is similar to that of rebuilding state discussed above, in that it facilitates saving only the information most critical to program state. The designer can exclude large working set arrays, string constants, and other similar memory areas from being checkpointed.
When these techniques are combined, the cost of checkpointing can be reduced by factors of 3-4. Checkpointing, like any fault tolerant computing technique, does require additional resources. Whether or not it will work well, is high dependant on both the target system design, and the application. Typically those systems which must meet hard real-time deadlines will have the most difficulty implementing any type of checkpoint-recovery system
RestoringExecutivestate
When a failure has occurred, the recovery mechanism restores system state to the last checkpointed value. This is the fundamental idea in the tolerance of a fault within a system employing checkpoint-recovery. Ideally, the state will be restored to a condition before the fault occurred within the system. After the state has been restored, the system can continue normal execution.
State is restored directly from the last complete snapshot, or reconstructed from the last snapshot and the incremental checkpoints. The concept is similar to that of a journaled file system, or even RCS(revision control system), in that only the changes to a file are recorded. Thus when the file is to be loaded or restored, the original document is loaded, and then the specified changes are made to it. In a similar fashion, when the state is restored to a system which has undergone one or more incremental checkpoints, the last full checkpoint is loaded, and then modified according to the state changes indicated by the incremental checkpoint data.
If the root cause of the failure did not manifest until after a checkpoint, and that cause is part of the state or input data, the restored system is likely to fail again. In such a case the error in the system may be latent through several checkpoint cycles. When the it finally activates and causes a system failure, the recovery mechanism will restore the state (including the error!) and execution will begin again, most likely triggering the same activation and failure. Thus it is in the system designers best interest to ensure that any checkpoint-recovery based system is fail fast - meaning errors are either tolerated, or case the system to fail immediately, with little or no incubation period.
Such recurring failures might be addressed through multi-level rollbacks and/or algorithmic diversity. Such a system would detect multiple failures as described above, and recover state from checkpoint data previous to the last recovery point. Additionally, when the system detects such multiple failures it might switch to a different algorithm to perform its functionality, which may not be susceptible to the same failure modes. The system might degrade its performance by using a more robust, but less efficient algorithm in an attempt to provide base level functionality to get past the fault before switching back to the more efficient routines.
Failure Detection
Failure detection can be a tricky part of any fault tolerant design. Sometimes the line between an unexpected (but correct) result, and garbage out is difficult to discern. In traditional checkpointrecovery
failure detection is somewhat simplistic. If the process or system terminates, there is a failure. Additionally, some systems will recover state if they attempted a non-transactional operation that failed and returned. The discussion of failure detection, and especially how it

impacts embedded systems is left to the chapters on fault tolerance, reliability, dependability, and architecture.

2 **Explain about the log based rollback recovery**.

Log based Recovery

Atomicity property of DBMS states that either all the operations of transactions must be performed or none. The modifications done by an aborted transaction should not be visible to database and the modifications done by committed transaction should be visible.

To achieve our goal of atomicity, user must first output to stable storage informat ion describing the modifications, without modifying the database itself. This information can help us ensure that all modifications performed by committed transactions are reflected in the database. This information can also help us ensure that no modifications made by an aborted transaction persist in the database.

Log and log records –

The log is a sequence of log records, recording all the update activities in the database. In a stable storage, logs for each transaction are maintained. Any operation which is performed on the database is recorded is on the log. Prior to performing any modification to database, an update log record is created to reflect that modification.

An update log record represented as: <Ti, Xj, V1, V2> has these fields:

1. **Transaction identifier:** Unique Identifier of the transaction that performed the write operation.
2. **Data item:** Unique identifier of the data item written.
3. **Old value:** Value of data item prior to write.
4. **New value:** Value of data item after write operation.

Other type of log records are:

1. <Ti start>: It contains information about when a transaction Ti starts.
2. <Ti commit>: It contains information about when a transaction Ti commits.
3. <Ti abort>: It contains information about when a transaction Ti aborts.

Undo and Redo Operations –

Because all database modifications must be preceded by creation of log record, the system has available both the old value prior to modification of data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

1. **Undo:** using a log record sets the data item specified in log record to old value.
2. **Redo:** using a log record sets the data item specified in log record to new value.

The database can be modified using two approaches –

1. **Deferred Modification Technique:** If the transaction does not modify the database until it has partially committed, it is said to use deferred modification technique.
2. **Immediate Modification Technique:** If database modification occur while transaction is still active, it is said to use immediate modification technique.

Recovery using Log records –

After a system crash has occurred, the system consults the log to determine which transactions need to be redone and which need to be undone.

1. Transaction Ti needs to be undone if the log contains the record <Ti start> but does not contain either the record <Ti commit> or the record <Ti abort>.
2. Transaction Ti needs to be redone if log contains record <Ti start> and either the record <Ti commit> or the record <Ti abort>.

Use of Checkpoints –

When a system crash occurs, user must consult the log. In principle, that need to search the entire log to determine this information. There are two major difficulties with this approach:
1. The search process is time-consuming.
2. Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will cause recovery to take longer.
To reduce these types of overhead, user introduce checkpoints. A log record of the form <checkpoint L> is used to represent a checkpoint in log where L is a list of transactions active at the time of the checkpoint. When a checkpoint log record is added to log all the transactions that have committed before this checkpoint have <Ti commit> log record before the checkpoint record. Any database modifications made by Ti is written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operation on Ti.
After a system crash has occurred, the system examines the log to find the last <checkpoint L> record. The redo or undo operations need to be applied only to transactions in L, and to all transactions that started execution after the record was written to the log. Let us denote this set of transactions as T. Same rules of undo and redo are applicable on T as mentioned in Recovery using Log records part.
Note that user need to only examine the part of the log starting with the last checkpoint log record to find the set of transactions T, and to find out whether a commit or abort record occurs in the log for each transaction in T. For example, consider the set of transactions {T0, T1, . . ., T100}. Suppose that the most recent checkpoint took place during the execution of transaction T67 and T69, while T68 and all transactions with subscripts lower than 67 completed before the checkpoint. Thus, only transactions T67, T69, . . ., T100 need to be considered during the recovery scheme. Each of them needs to be redone if it has completed (that is, either committed or aborted); otherwise, it was incomplete, and needs to be undone.

UNIT-V
PART-A

1 **What is peer to peer system?**
Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the
Internet and other networks in ever-increasing numbers.

2 **What is goal of peer to peer system?**
The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for separately managed servers and their associated infrastructure.

3 **What are the characteristics of peer to peer system? MAY/JUNE 2016**
Their design ensures that each user contributes
resources to the system.
• Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
• Their correct operation does not depend on the existence of any centrally administered systems.
• They can be designed to offer a limited degree of anonymity to the providers and

users of resources.

5 **What is the need of peer to peer middleware system? NOV/DEC 2017**

Peer-to-peer middleware systems are designed specifically to meet the need for the automatic placement and subsequent location of the distributed objects managed by peer-to-peer systems and applications.

6 **Write the Non-functional requirements of peer-to-peer middleware system?**

- Global scalability
- Load balancing
- Optimization for local interactions between neighbouring peers
- Accommodating to highly dynamic host availability

7 **What is the role of routing overlays in peer to peer system? APR/MAY 2017**

Peer-to-peer systems usually store multiple
replicas of objects to ensure availability. In that case, the routing overlay maintains Knowledge of the location of all the available replicas and delivers requests to the nearest 'live' node (i.e. one that has not failed) that has a copy of the relevant object.

8

What are the tasks performed by routing overlay?

- Insertion of objects
- Deletion of objects
- Node addition and removal

9 **What are the generations of peer to peer system?**

Three generations of peer-to-peer system and application development can be identified.

- The first generation was launched by the Napster music exchange service [OpenNap 2001].
- A second generation of file sharing applications offering greater scalability, anonymity and fault tolerance quickly followed including Freenet, Gnutella, Kazaa and BitTorrent
- The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed
- resources on a global scale

10 **What are the case studies used in overlay?**
NOV/DEC 2017

- **Pastry** is the message routing infrastructure deployed in several applications including PAST.
- **Tapestry** is the basis for the Ocean Store storage system.

11. **Give the characteristics of Peer-to-Peer systems? JUNE 2016, NOV 2017, APRIL/MAY 2018**

Its design ensures that each user contributes resources to the system.
Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
Its correct operation does not depend on the existence of any centrally administered systems.
They can be designed to offer a limited degree of anonymity to the providers and users of resources.

A key issue for their efficient operation is the choice of an algorithm for the placement of data across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads.

PART-B

1 **With neat sketch explain Routing Overlays in detail. MAY/JUNE 2016, NOV/DEC 2016,APRIL/MAY 2017, APRIL/MAY 2018**

Resilient Overlay Networks (RON), an architecture that allows end-to-end communication across the wide-area Internet to detect and recover from path outages and periods of degraded performance within several seconds. A RON is an application-layer overlay on top of the existing Internet routing substrate. The overlay nodes monitor the liveliness and quality of the Internets paths among themselves, and they use this information to decide whether to route packets directly over the Internet or by way of the The RON nodes monitor the functioning and quality of the Internet paths among themselves, and use this information to decide whether to route packets directly over the Internet or by way of other RON nodes, optimizing applicationspecific
routing metrics.

- build on the top of another network such as ATM etc.
- IP itself id build on the top of another network.
- The term usually means a network on the top of IP.

Motivation behind Overlay Networks:-

Internet suffers from following four important drawbacks:

1. *slow link failure recovery :* BGP takes a long time, of the order of several minutes, to converge to a new valid route after a router or link failure causes a path outage.
2. *Inability to detect path or performance failure:* BGP cannot detect many problems like floods, persistent congestion, etc. that can greatly affects the performance. As long as a link is deemed "live" i.e.. the BGP session is still alive, BGP's AS-path-based routing will continue to route packets down the fault path.
3. *Inability to effectively multi-home end-customer networks:* As "solution" to Internet unreliability (Instability) is to multi-homing. Unfortunately, peering at the network level by small customers break down wide-area routing scalability.
4. *Blunt policy expression:* BGP is unable of expressing fine-grained policies aimed at users or remote hosts; it can only express policies at the granularity of entire remote networks. This reduces the set of paths in the case of failures.

RON overcome this drawbacks of BGP.

A RON Model:-

- Designate RON nodes for the overlay.
- Exchange of performance and reach ability, and routing based on this.
- 2-50 nodes (only) on overlay.

The RON architecture achieves the following benefits:

1. *Fault detection:* A RON can more efficiently find alternate paths around problems even when the underlying network layer incorrectly believes that all is well.
2. *Better reliability for applications:* Each RON can have an independent, application-specific definition of what constitutes a fault.
3. *Better performance:* A RON's limited size allows it to use more aggressive path computation algorithms that the Internet. RON nodes can exchange more complete topologies, collect more

detailed link quality metrics, execute more complex routing algorithms , and respond more quickly to change.
4. *Application-specific routing:* Distributed applications can link with the RON library and choose, or even define, their own routing matrices.
• Software modules at RON node look into the following
- RON client
- Routing
- Data Forwarding
- Bootstrap and Membership management
- Link state based dissemination
- Monitoring Virtual Links.
- Path-Evaluation and Selection
• Full mesh network among members.

Possible Usage Models:-

• A specific application (like Video conferencing) construct and uses RON.
• A network administrator construct an overlay.
• Overlay ISP.

Failure Detection in RON:-

• Uses UDP heartbeat packet
- Failure detection in Overlay is application specific. In multimedia conferencing 5% loss rate may bark the video whereas a FTP application can still work with lower throughput.
- But one cannot reduce heart beat interval to a very small value. That will give rise to false alarm.
- Also there is a trade off between overhead vs. detection time.

Matrices:-

• Latency
- RON expects reply of heart beat from which it calculates RTT.
- RTTs are stable over of the order of 15 mints to 1 hr.
- If spikes occur in the middle, then that will be smoothen out by EWMA.
• Packet Loss Rate
- Simply use heart beat and from this measure loss loss rate.
- if p_1, p_2 are the loss rate of link1 and link2 respectively, then the loss rate of the path using consisting og link1 and link2 is $1-(1-p_1)(1-p_2)$.
.