1.1 Introduction

Networks of computers are everywhere. The Internet is one, as are the many networks of which it is composed. Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks, all of these, both separately and in combination, share the essential characteristics that make them relevant subjects for study under the heading *distributed systems*. In this book we aim to explain the characteristics of networked computers that impact system designers and implementors and to present the main concepts and techniques that have been developed to help in the tasks of designing and implementing systems that are based on them.

We define a distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or the same room. Our definition of distributed systems has the following significant consequences:

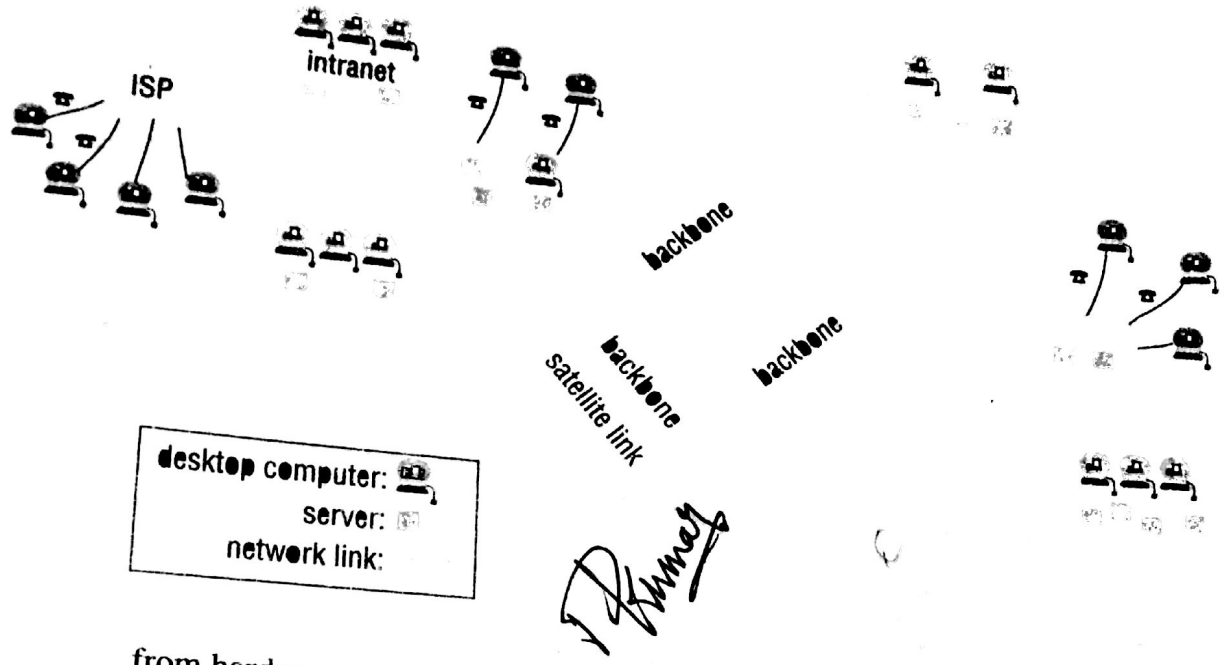
Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network. We will describe ways in which this extra capacity can be usefully deployed at many points in this book. The coordination of concurrently executing programs that share resources is also an important and recurring topic.

No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network. Examples of these timing problems and solutions to them will be described in Chapter 11.

Independent failures: All computer systems can fail and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*) is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running. The consequences of this characteristic of distributed systems will be a recurring theme throughout the book.

The motivation for constructing and using distributed systems stems from a desire to share resources. The term 'resource' is a rather abstract one, but it best characterizes the range of things that can usefully be shared in a networked computer system. It extends

Figure 1.1 A typical portion of the Internet



from hardware components such as disks and printers to software-defined entities such as files, databases and data objects of all kinds. It includes the stream of video frames that emerges from a digital video camera and the audio connection that a mobile phone call represents.

The purpose of this chapter is to convey a clear view of the nature of distributed systems and the challenges that must be addressed in order to ensure that they are successful. Section 1.2 gives some key examples of distributed systems, the components from which they are constructed and their purposes. Section 1.3 explores the design of resource-sharing systems in the context of the World Wide Web. Section 1.4 describes the key challenges faced by the designers of distributed systems: heterogeneity, openness, security, scalability, failure handling, concurrency and the need for transparency.

1.2 Examples of distributed systems

Our examples are based on familiar and widely used computer networks: the Internet, intranets and the emerging technology of networks based on mobile devices. They are designed to exemplify the wide range of services and applications that are supported by computer networks and to begin the discussion of the technical issues that underlie their implementation.

1.2.1 The Internet

The Internet is a vast interconnected collection of computer networks of many different types. Figure 1.1 illustrates a typical portion of the Internet. Programs running on the

1.2.3 Mobile and ubiquitous computing

Technological advances in device miniaturization and wireless networking have led increasingly to the integration of small and portable computing devices into distributed systems. These devices include:

- Laptop computers.
- Handheld devices, including personal digital assistants (PDAs), mobile phones, pagers, video cameras and digital cameras.
- Wearable devices, such as smart watches with functionality similar to a PDA.
- Devices embedded in appliances such as washing machines, hi-fi systems, cars and refrigerators.

The portability of many of these devices, together with their ability to connect conveniently to networks in different places, makes *mobile computing* possible. Mobile computing (also called *nomadic computing* [Kleinrock 1997]) is the performance of computing tasks while the user is on the move, or visiting places other than their usual environment. In mobile computing, users who are away from their 'home' intranet (the intranet at work, or their residence) are still provided with access to resources via the devices they carry with them. They can continue to access the Internet; they can continue to access resources in their home intranet; and there is increasing provision for users to utilize resources such as printers that are conveniently nearby as they move around. The latter is also known as *location-aware* or *context-aware computing*.

Ubiquitous computing [Weiser 1993] is the harnessing of many small, cheap computational devices that are present in users' physical environments, including the home, office and even natural settings. The term 'ubiquitous' is intended to suggest that small computing devices will eventually become so pervasive in everyday objects that they are scarcely noticed. That is, their computational behaviour will be transparently and intimately tied up with their physical function.

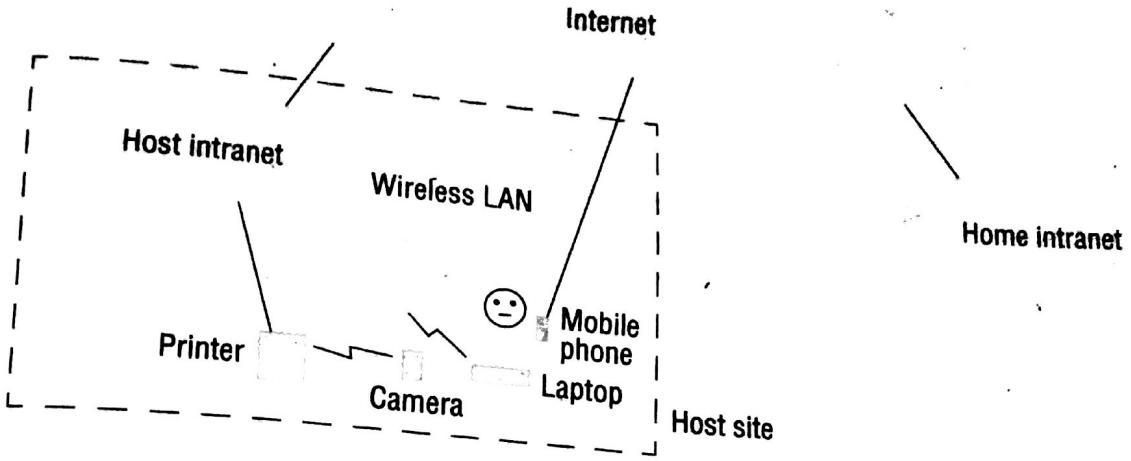
The presence of computers everywhere only becomes useful when they can communicate with one another. For example, it would be convenient for users to control their washing machine and their hi-fi system from a 'universal remote control' device in the home. Equally, the washing machine could page the user via a smart badge or watch when the washing is done.

Ubiquitous and mobile computing overlap, since the mobile user can in principle benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous computing could benefit users while they remain in a single environment such as the home or a hospital. Similarly, mobile computing has advantages even if it involves only conventional, discrete computers and devices such as laptops and printers.

Figure 1.3 shows a user who is visiting a host organization. The figure shows the user's home intranet and the host intranet at the site that the user is visiting. Both intranets are connected to the rest of the Internet.

The user has access to three forms of wireless connection. Their laptop has a means of connecting to the host's wireless LAN. This network provides coverage of a few hundreds of metres (a floor of a building, say). It connects to the rest of the host intranet via a gateway. The user also has a mobile (cellular) telephone, which is connected to the Internet. The phone gives access to pages of simple information, which

Figure 1.3 Portable and handheld devices in a distributed system



it presents on its small display. Finally, the user carries a digital camera, which can communicate over a personal area wireless network (with range up to about 10m) with a device such as a printer.

With a suitable system infrastructure, the user can perform some simple tasks in the host site using the devices they carry. While journeying to the host site, the user can fetch the latest stock prices from a web server using the mobile phone. During the meeting with their hosts, the user can show them a recent photograph by sending it from the digital camera directly to a suitably enabled printer in the meeting room. This requires only the wireless link between the camera and printer. And they can in principle send a document from their laptop to the same printer, utilizing the wireless LAN and wired Ethernet links to the printer.

Mobile and ubiquitous computing are a lively area of research and they are the subject of Chapter 16.

1.3 Resource sharing and the Web

Users are so accustomed to the benefits of resource sharing that they may easily overlook their significance. We routinely share hardware resources such as printers, data resources such as files, and resources with more specific functionality such as search engines.

Looked at from the point of view of hardware provision, we share equipment such as printers and disks to reduce costs. But of far greater significance to users is their sharing of the higher-level resources that play a part in their applications and in their everyday work and social activities. For example, users are concerned with sharing data in the form of a shared database or a set of web pages – not the disks and processors that those are implemented on. Similarly, users think in terms of shared resources such as a search engine or a currency converter, without regard for the server or servers that provide these.

1.4 Challenges

The examples in Section 1.2 are intended to illustrate the scope of distributed systems and to suggest the issues that arise in their design. Although distributed systems are to be found everywhere, their design is quite simple and there is still a lot of scope to develop more ambitious services and applications. Many of the challenges discussed in this section have already been met, but future designers need to be aware of them and to be careful to take them into account.

1.4.1 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:

- networks;
- computer hardware;
- operating systems;
- programming languages;
- implementations by different developers.

Although the Internet consists of many different sorts of network (illustrated in Figure 1.1), their differences are masked by the fact that all of the computers attached to them use the Internet protocols to communicate with one another. For example, a computer attached to an Ethernet has an implementation of the Internet protocols over the Ethernet, whereas a computer on a different sort of network will need an implementation of the Internet protocols for that network. Chapter 3 explains how the Internet protocols are implemented over a variety of different networks.

Data types such as integers may be represented in different ways on different sorts of hardware for example, there are two alternatives for the byte ordering of integers. These differences in representation must be dealt with if messages are to be exchanged between programs running on different hardware.

Although the operating systems of all computers on the Internet need to include an implementation of the Internet protocols, they do not necessarily all provide the same application programming interface to these protocols. For example, the calls for exchanging messages in UNIX are different from the calls in Windows.

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. The Common Object Request Broker (CORBA), which is described in Chapters 4, 5 and 20, is an example. Some middleware, such as Java Remote Method Invocation (RMI) (see Chapter 5) supports only a single programming language. Most middleware is implemented over the Internet protocols, which themselves mask the difference of the underlying networks. But all middleware deals with the differences in operating systems and hardware – how this is done is the main topic of Chapter 4.

In addition to solving the problems of heterogeneity, middleware provides a uniform computational model for use by the programmers of servers and distributed applications. Possible models include remote object invocation, remote event notification, remote SQL access and distributed transaction processing. For example, CORBA provides remote object invocation, which allows an object in a program running on one computer to invoke a method of an object in a program running on another computer. Its implementation hides the fact that messages are passed over a network in order to send the invocation request and its reply.

Heterogeneity and mobile code ◊ The term *mobile code* is used to refer to code that can be sent 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. For example, executable files sent as email attachments by Windows/x86 users will not run on an x86 computer running Linux or a Macintosh computer running Mac OS X.

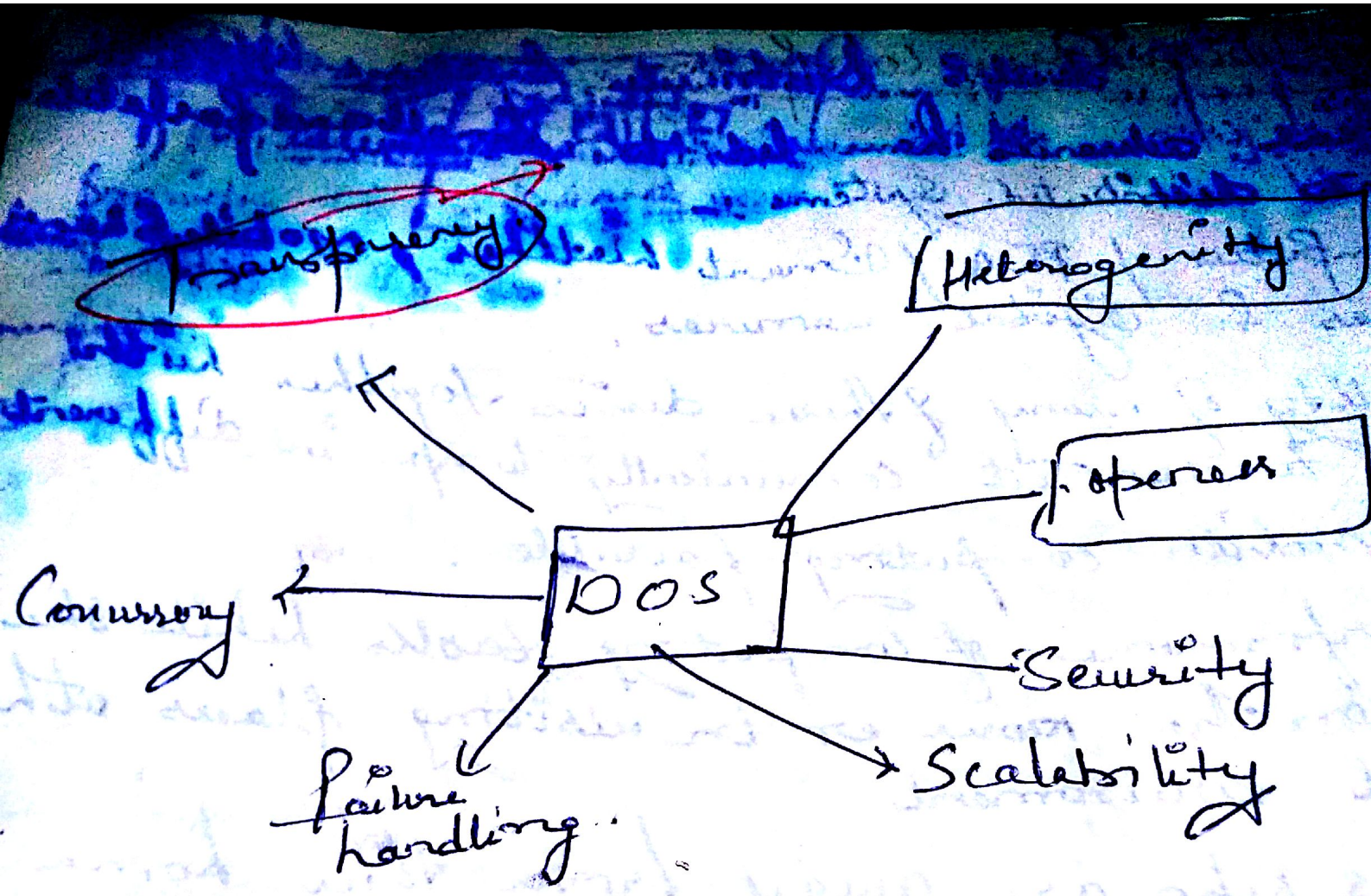
The *virtual machine* approach provides a way of making code executable on any hardware: the compiler for a particular language generates code for a virtual machine instead of a particular hardware order code for example, the Java compiler produces code for the Java virtual machine, which needs to be implemented once for each type of hardware to enable Java programs to run. However, the Java solution is not generally applicable to programs written in other languages.

1.4.2 Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented 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.

Openness cannot be achieved unless the specification and documentation of the key software interfaces of the components of a system are made available to software developers. In a word, the key interfaces are *published*. This process is akin to the standardization of interfaces, but it often bypasses official standardization procedures, which are usually cumbersome and slow-moving.

However, the publication of interfaces is only the starting point for adding and extending services in a distributed system. The challenge to designers is to tackle the complexity of distributed systems consisting of many components engineered by different people.



deployment.
→ Most web applications are three tier.

Applications :-

1) Telecommunication n/w
   ↳ telephone n/w & cellular n/w
   ↳ computer n/w such as Internet
   ↳ Wireless Sensor n/w's
   ↳ Routing algorithm.

2) n/w applications :
   ↳ World wide web & peer to peer n/w's
   ↳ massively multiplayer online games & virtual
   ↳ distributed databases.
   ↳ n/w file systems.

3) Real time process control :
   aircraft control sys

1) Parallel computation / Banking System
   → Scientific

→ Advantages of DS

1) Sharing data :- There is a provision in the one site may be able to access data

2) Autonomy :-

3) Comm^n of flexibility

4)

↳ Rollback :- It involves the design of sys so that the state of permanent data can be restored or rollback after server has crashed.

6) Concurrency :-

7) Transparency :- ⟵ designer must hide the complexity of the system as much as they can

Transparency is defined as the concealment from the user & the application programmer of the separation of components in a DS, so that the system is perceived as a whole rather than a collection of independent components.

- Access transparency :- Enables local & remote resources to be accessed using identical operations
- Local transparency :- enables resources to be accessed without knowledge of their physical location
- Concurrency :- enables several processes to operate concurrently using shared resources w/o interference between them
- Replication :- Enable multiple instances of resources used to increase reliability & performance.

Failure T :- allowing users & application programs to complete their tasks despite the failure of HW or SW components.

Mobility transparency :- allows the movement of resources & clients within a system without affecting the operation of users.

Performance T :- allows the system to be reconfigured to improve performance as loads vary

(Theoritical foundations :-)

Limitation of a Global clock in DS

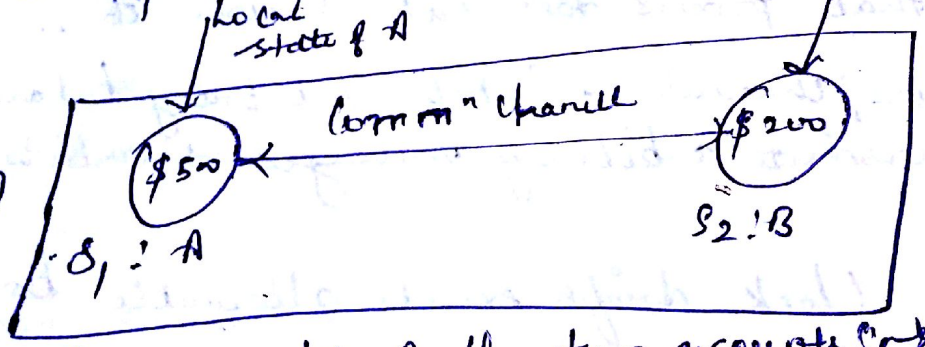
1) No Global clock :- Because of the absence of a Global clock in a distributed system, obtaining a coherent global state of the system is difficult.

(if all the observations of different processes are made at the same physical time)

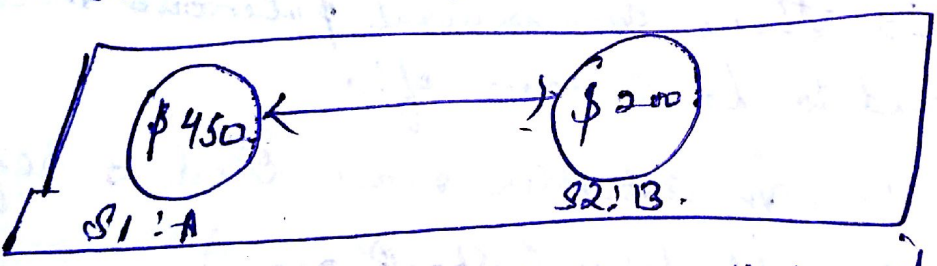
following eg :- illustrates the difficulty in obtaining a Coherent Global state :

→ 1) Let S_1 & S_2 are two distint entities, that maintain Paytm accounts A & B respectively, where A & B can be refer as process.

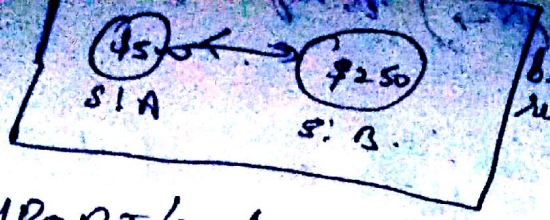
2) Knowledge of the global state of the system may be necessary to compute the net balance of both accounts.



Initial state of the two accounts



the collection of a Global state, if state S_2 records the st immediately after the debit has occured & S_2 save the of B before the fund transfer message has reache the Global state of the system shows $50 is



if A's state is recorded before the transfer & B's state recorded after a message has been sent

LAMPORT'S LOGICAL CLOCK

Lamport's proposed a Scheme to order events in a dis using logi clocks.

→ The execution of processes is characterized by a sequence of events

eg: Sending of a message constitutes one event, receiving → a

Def^n :→ Due to absence of a global clock, the order of the i which two events occur at two different computers cannot be determined based on the local time.

→ However, it is possible to ascertain the order in which two e occur based solely on the behaviour exhibited by the underlyin computation.

HAPPENED BEFORE RELATION (→) :→ It captures the causal dependencies b/w pairs of events i.e whether two events are causall related or not. The relation (→) is defined as :-

1) $a \to b$, if a & b are two events in the same process & a occured before b.

2) $a \to b$, if a is the event of sending message 'm' in a pro & b is the " " receiving " same message another process !

3) If $a \to b$ & $b \to c$, then $a \to c$, then " → " is transitiv

Note: In DS, processes interact with each other & affect outcome of event of processes.

→ In general, an events changes the system, which in turn influences the occurence & outcome of future events

That is, past events influence future events and this influence among causally related events (those events

CAUSALLY RELATED EVENTS :→

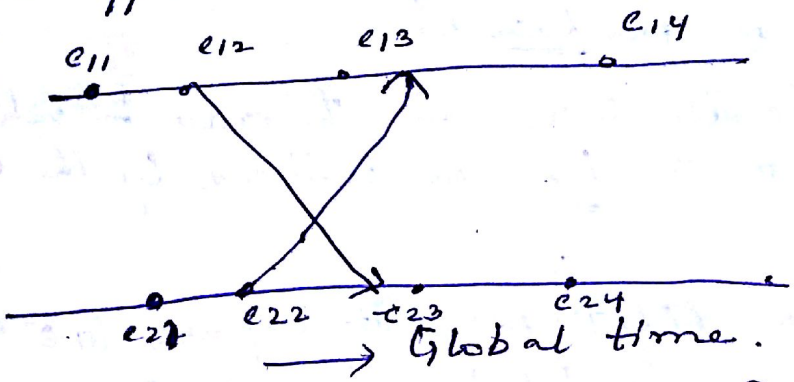
Event 'a' causally affects event 'b' if $a \rightarrow b$.

CONCURRENTLY EVENTS :-

Two distinct events a and b are said to be concurrent (denoted by $a \| b$) if $a \not\rightarrow b$ and $b \not\rightarrow a$. In other words concurrents, concurrent events do not causally affect each other.

for any two events a and b in a system, either $a \rightarrow b$, $b \rightarrow a$ or $a \| b$.

Eg:



$e_{11}, e_{12}, e_{13}, e_{14}$ are events in P_1

P_2

The arrow represent message transfer b/w the processes.

) e_{12} is the event of sending the message at P_1 & e_{23} is the event of receiving the same message at P_2.

g:→ CAUSALLY related Events

↳ $e_{22} \rightarrow e_{13}$, $e_{13} \rightarrow e_{14}$ & therefore

$e_{22} \rightarrow e_{14}$.

this means e_{22} causally affects event e_{14}.

TE Whenever $a \rightarrow b$ holds for two events a and b
= there exists a path which moves only forward the time axis.

CONCURRENT EVENTS :→

e_{21} & e_{11} are concurrent event though e_{11} appears to have occurred before e_{21} in real (global) time for a global observer.

LOGICAL CLOCKS :→

In order to realize the relation →, Lamport introduced the following system of logical clocks.

→ There is a clock C_i at each process P_i in the system.

→ The clock C_i can be thought of as a function that assigns a number $C_i(a)$ to any event a, called the timestamp of event a at P_i.

→ The no assigned by the system of clocks have no relation to physical time, & hence name Logical Clocks.

→ The logical clocks take monotonically increasing values.

→ These clocks can be implemented by counters.

CONDITION SATISFIED BY THE SYSTEM OF CLOCKS

For any two event a & b :

if $a \rightarrow b$ then $C(a) \rightarrow C(b)$.

This happened before relation ($\rightarrow$) can now be realized by using the logical clocks. If the two conditions are

1) for any two events 'a & b' in a process P_i, if 'a' occurs before 'b', then $C_i(a) < C_i(b)$

2) If a is the event of sending message 'm' in process P_i & 'b' is the " " receiving same "m" in process P_j then $C_i(a) < C_j(b)$.

IMPLEMENTATION RULES

that the clocks satisfy the conditions

IR1) Clock C_i is implemented b/w any two successive events in process P_i $C_i = C_i + d$ $(d > 0)$

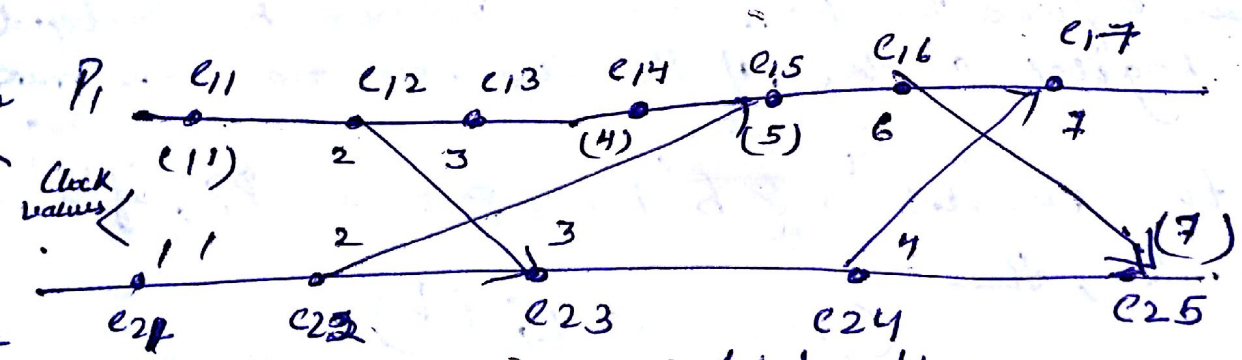
If $a \& b$ are two successive events in P_i and $a \to b$, then $C_i(b) = C_i(a) + d$.

IR2) If event a is the sending of message m by process P_i, then message m is assigned a timestamp $tm = C_i(a)$

→ On receiving the same message 'm' by process P_j, C_j is set to a value greater than or equal to it's present value & greater than tm.

$C_j = \max(C_j, tm + d)$ $d > 0$

EX: HOW LOGICAL CLOCKS are UPDATED UNDER LAMPORT'S ~~SEEMEN~~ SCHEME,



1) Both the clock values CP_1 & CP_2 are assumed to initially & d is assumed to be 1.

) e_{11} is an internal event in process P_1 which c

$e_{11} = 1$, $CP_1 = 1$

e_{21} & e_{22} are two events in P_2, $CP_2 = 2$.

The event e_{25}, corresponding to the receive event message, increments the clock CP_2 to 7

$(\max(4+1, 6+1))$

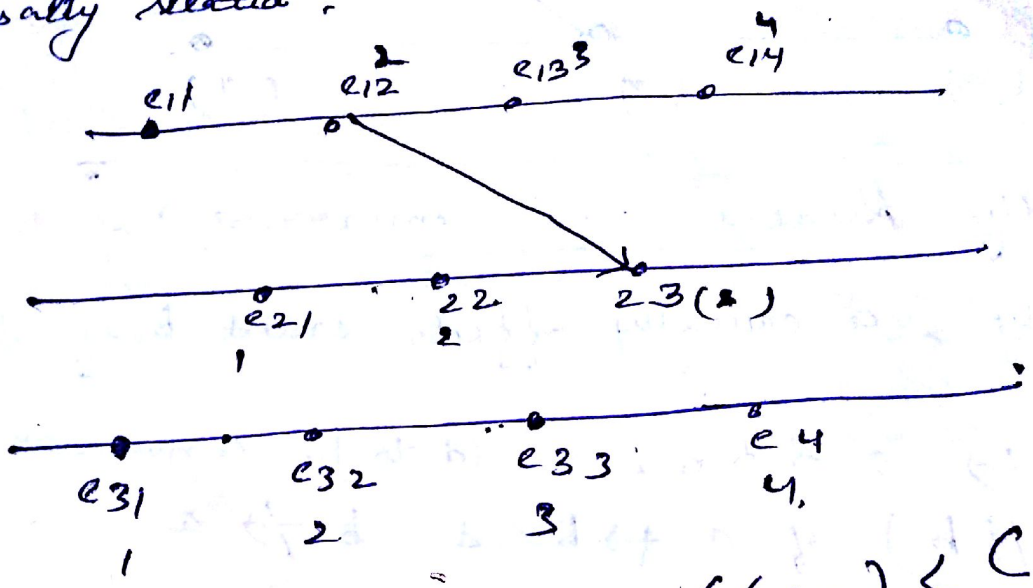
LIMITATION OF LAMPORT'S CLOCK

NOTE, In Lamport's system of logical clocks, if $a \rightarrow b$ then $C(a) < C(b)$. However the reverse is not necessarily true if the events have occurred in different processes.

↳ Means, if $C(a) < C(b)$, then $a \rightarrow b$ is not necessarily true.

→ Events a and b may be causally related or may not be causally related.

Eg:

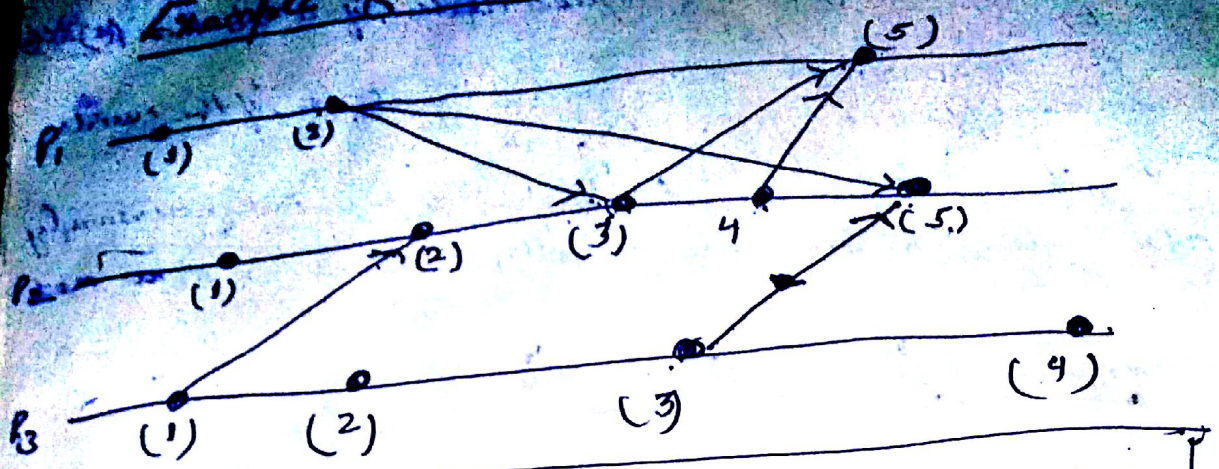


(eg $C(e_{11}) < C(e_{23})$ & $C(e_{11}) < C(e_{32})$.

However, we can see from the figure that event e_{11} is causally related to e_{23}, but not to event e_{32}.

NOTE) USING the timestamps assigned by Lamport clock, we cannot reason about the causal relationship b/w two events occuring in different processes by looking at the timestamps of the events.

Example of Lamport Logical clock

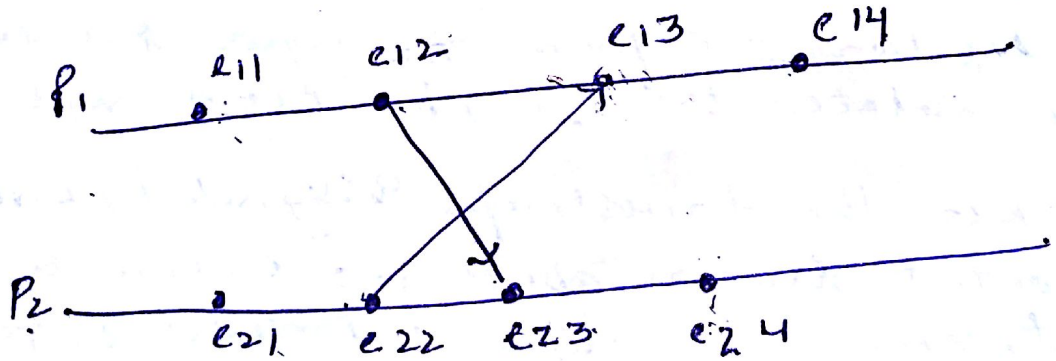


Q:- Causally Related vs. Concurrent :-

1) Casually, a causally affects event b if $a \rightarrow b$

Concurrent :- a & b are said to be concurrent (denoted by $a \| b$) if $a \not\rightarrow b$ & $b \not\rightarrow a$

Q:



e_{11} & e_{21} are concurrent events

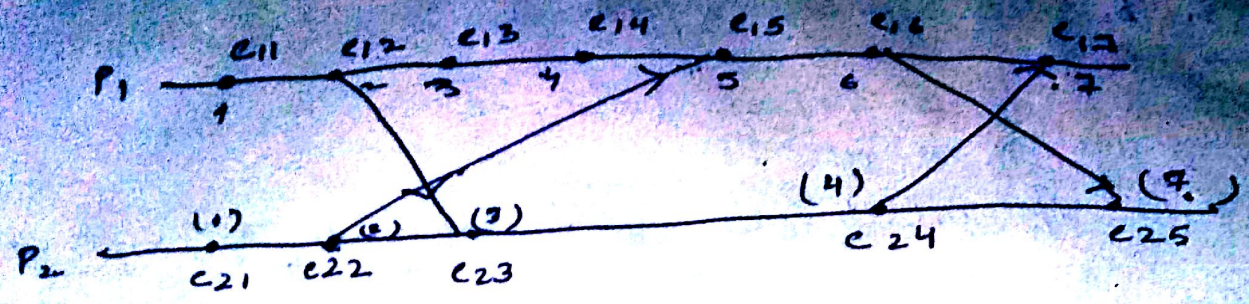
e_{14} & e_{23} "

e_{22} casually affects e_{14}.

Note: Concurrent events :- If a and b occur on different processes that do not exchange messages, then neither $a \rightarrow b$ nor $b \rightarrow a$ are true.

→

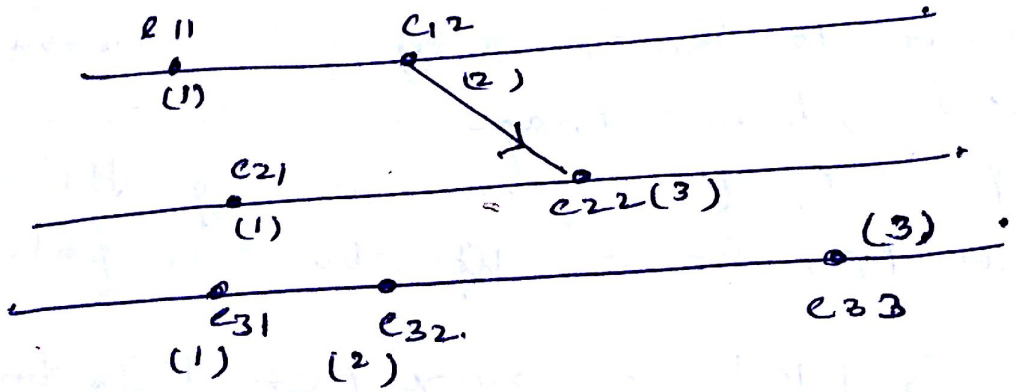
LAMPORT's Logical Clock



Limitation of LAMPORT's CLOCK :→ $a \rightarrow b$

LOGICAL CLOCK :→ Event a happens before event b ⇒ the logical clock value of a is smaller than the logical clock value of b. ($C(a) < C(b)$).

eg:



eg :→ $C(e_{11}) < C(e_{22})$ and $e_{11} \rightarrow e_{22}$ is

$C(e_{11}) < C(e_{32})$ but $e_{11} \rightarrow e_{32}$ is

* Cannot determine whether two events are casually from timestamp.

Vector clock :- The system of vector clocks was developed independently by Fidge and Mattern.

→ Let n be the number of processes in a distributed system.

→ Each process P_i is equipped with a clock C_i, which is an integer vector of length n.

→ $C_i(a)$ is referred to as the timestamp of event a at P_i.

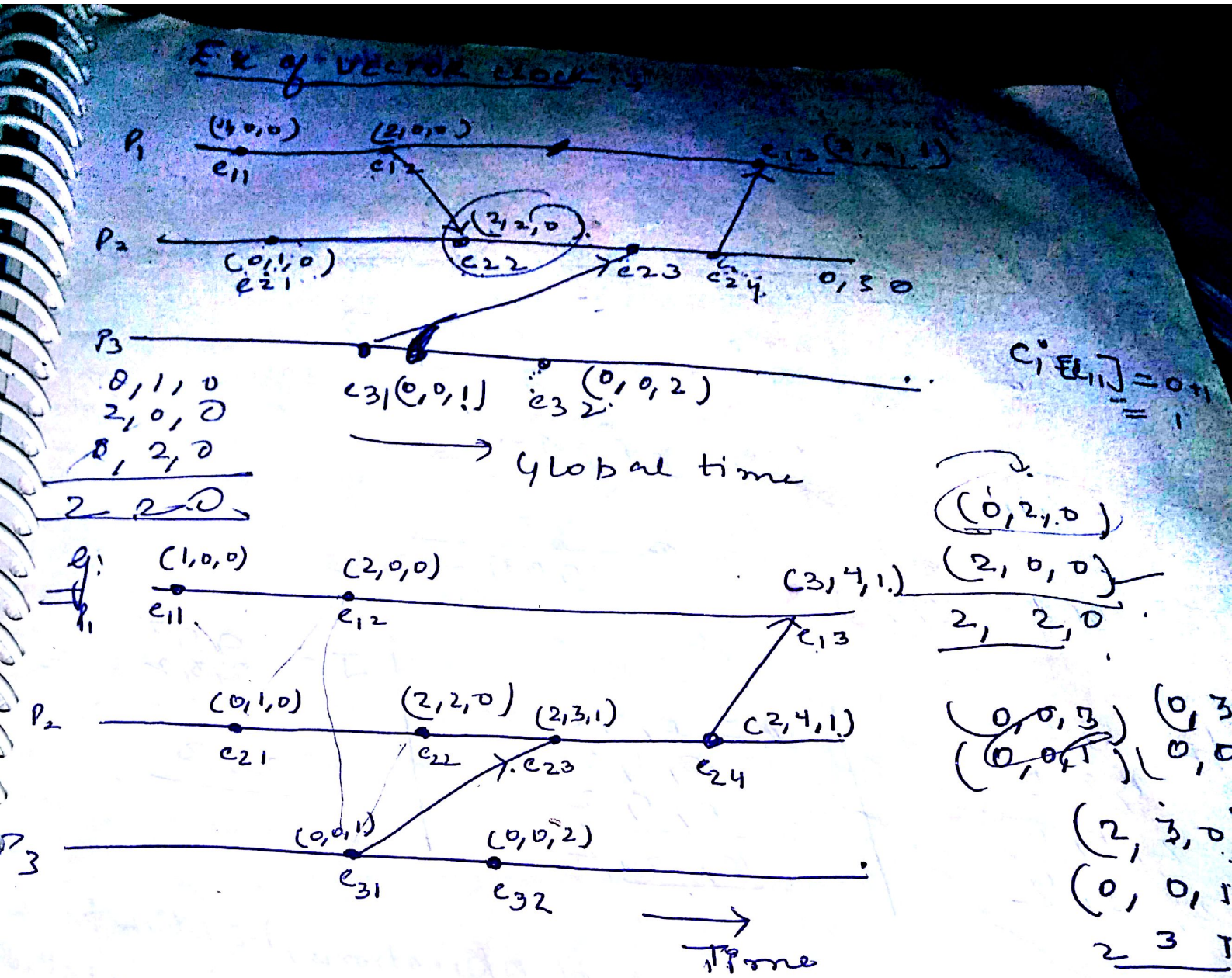
IR for the vectors clocks are as follows

IR1 Clock C_i is incremented between any two successive events in process P_i.

$$C_i[i] = C_i[i] + d \quad [d \neq 1]$$

IR2 If event a is the sending of the message m by process P_i, then message m is assigned a vector timestamp $t_m = C_i(a)$; on receiving the same m by processes P_J, C_J is updated as follows:-

$$\forall K \quad C_J[K] := \max(C_J[K], t_m[K]$$



→ Thus, the System of vector clocks allows order event and decide whether two event casually related or not by simply looking timestamps of the events.