Distributed Systems

International Institute of Information Technology

Hyderabad, India

Pictures not cited explicitly are taken
from the course book or wikepedia.

Course Policies

Textbook(s)

- Distributed Computing: Principles, Algorithms, and Systems: Kshemkalyani and Singhal.
- Distributed Algorithms: Gerald Tel, Nancy Lynch

Grading policy – Almost Final!

- Slip Tests: 15
- Project:20
- Assignments : 20 (best 2 of 3 taken): 2 coding and one theory
- 1 Quiz + Midsem + Endsem: (10+12)+23=45

Syllabus (Tentative)

- Module 1
  - Introduction
  - Time and Synchronization
  - Mutual Exclusion
  - Deadlock Detection

- Module 2
  - Distributed graph algorithms
  - Consensus, Agreement, Locking
  - Deadlock Handling
  - Distributed File systems

Syllabus (Tentative)

- Module 4
  - Limitations of distributed computing
  - Self-Stabilization
  - CAP Theorem
  - Block Chain and BitCoins (Guest lecture)

TAs

- Tanish Lad
- Manish
- Akshat Goel
- R Guru Ravi Shanker
- Gadela Kesav

Assignments - Akshat
Slip Tests - Kesav
Quiz/Exams - Tanish and Manish
Project - Guru

Motivation

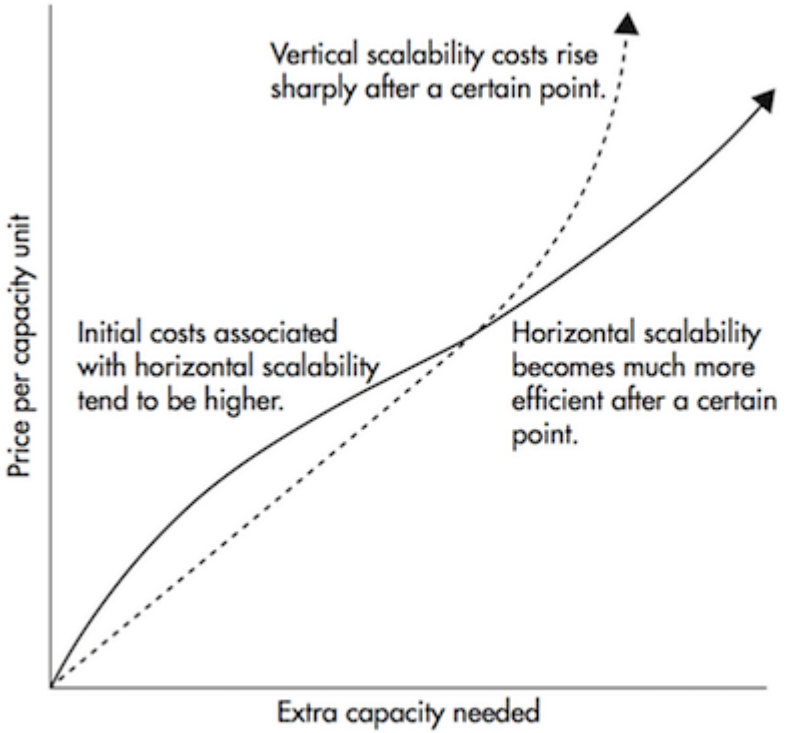
A group of computers working together as to appear as a single computer to the end-user.– Tannenbaum
Eg: Airline reservation, Online Banking

Required Features:
Lot of computers
Perform Concurrently
Fail independently
Dont share a global clock

Why Distributed Systems?
Best hardware may be insufficient to scale vertically (upgrade hardware of a system) hence the need to scale horizontally. Other benefits: ***Fault tolerance*** and ***low latency***.

Motivation

- Horizontal Scaling to fit data, shift data to manage load - scalability
- Inherently distributed. Ex: Banking system across cities
- Data Sharing. Ex Libraries, certain files, Complete datasets
- Enhanced Reliability
- Increased Performance/cost ratio (parallelistion)
- Fault Tolerance



Horizontal scaling becomes **much cheaper** after a certain threshold

Challenges with a Distributed System

- Unreliable communication – messages may drop, could be intercepted, could be delayed, could be out of order.
- Lack of global knowledge – a participating node has information of only his local memory and local computations. (no shared memory)
- Lack of synchronisation (different local clocks)-how to implement sequence in which we want things to happen; how to order steps across participating nodes where each node has a different clock
- Concurrency control – Data distributed across nodes. Moreover each has its own clock. How do you implement Mutual exclusion now or critical sections now?
- Failure and recovery : Nodes may fail, communication channels may fail – how do you still make sure the system functions. When a node recovers how do you make the updates on it.
- Deadlocks, Termination Detection, File systems are other concerns

Can we solve these -

The village without a mirror – consensus

War Preparation with messengers

- Consensus: All non-faulty processes must agree on a common value.
  If initial value of all non-faulty processes is *v*, then the agreed upon value must be *v*

Consensus with traitors included - the general too may be malicious (Byzantine Agreement Problem)

Time in Distributed Systems

Why is Time a Challenge?

IIIT Organises a coding competition. First prize to the first person who sends in the working code that runs on all test cases first and so on.

Raghav sent it at time t1 and it reached me at time t2
Sunita sent it at t3 and it reached me at t4.

Now who is first ?
Obviously t2< t4 does not imply t1<t3

Why is Time a Challenge?

IIIT Organises a coding competition. First prize to the first person who sends in the working code that runs on all test cases first and so on.

Raghav sent it at time t1 and it reached me at time t2
Sunita sent it at t3 and it reached me at t4.

Now who is first ?
Obviously t2< t4 does not imply t1<t3

Make sure Raghav and Sunita have their clocks set to the same time

Make sure both work with reference to a global time

Design something that inspite of their local clocks; I can still figure out who sent it first

Why is Time a Challenge?

Another example:

Consider the need to execute Mutual exclusion so that a banking transaction completes. A starts to transfer 1lakh to B and node B needs to know A+B.
Hence I need to apply the locks/semaphores.

How do I order these events such that B's query to know A+B does not get executed
midway A sending and B receiving.

This Lecture

- Understand the notion of time in distributed systems.
- How to maintain clocks in a distributed system.
  - Physical vs. Logical time
- Various algorithms/protocols for maintaining logical time.
- Snapshots and their uses
  - How to take a snapshot? Consistency of a snapshot.

Material and slides from Chapters 2 and 3 of the book.

Time in Distributed Systems

- Recall the quartz oscillation based clock ticks used in digital systems.
  - These clocks drift.
- With such a drift, it is not possible to arrange events according to a causal order.

Why is time important

- to determining the order in which events occur
- helps ensure liveness and fairness in mutual exclusion algorithms
- helps maintain consistency in replicated databases,
- helps design correct deadlock detection algorithms to avoid phantom and undetected deadlocks.
- Tracking of dependent events
- In debugging, it helps construct a consistent state
- for resuming reexecution; in failure recovery, it helps build a checkpoint;
- in replicated databases,
- it aids in the detection of file inconsistencies in case of a network partitioning

Motivation

- To ensure a correct and consistent view of a distributed database.
- Sensor networks and sensing properties of the physical world.
- Applications using time-outs to deduce certain actions/events.

Physical Time and Synchronization

- In centralized systems, there is only single clock. A process gets the time by simply issuing a system call to the kernel.
  - The time returned by the kernel is the “time”.
- In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time.
- These clocks can easily drift seconds per day, accumulating significant errors over time.
- This clearly poses serious problems to applications that depend on a synchronized notion of time.

Cristian's (Time Server) Algorithm

- Uses a *time server* to synchronize clocks
  - Time server keeps the reference time (say UTC)
  - A client asks the time server for time, the server responds with its current time, and the client uses the received value *T* to set its clock
- But network round-trip time introduces errors…
  - Let **RTT = response-received-time – request-sent-time** (measurable at client),
  - If we know (a) min = minimum client-server one-way transmission time and (b) that the server timestamped the message at the last possible instant before sending it back
  - Then, the actual time could be between **[T+min,T+RTT— min]** where RTT-min is the max time it took for the response to return which happens when request took min time to go

Berkeley UNIX algorithm

- One master machine and set of slave nodes
- Periodically, this daemon polls and asks all the machines for their time
- The machines respond.
- The daemon computes an average time and then broadcasts this average time.

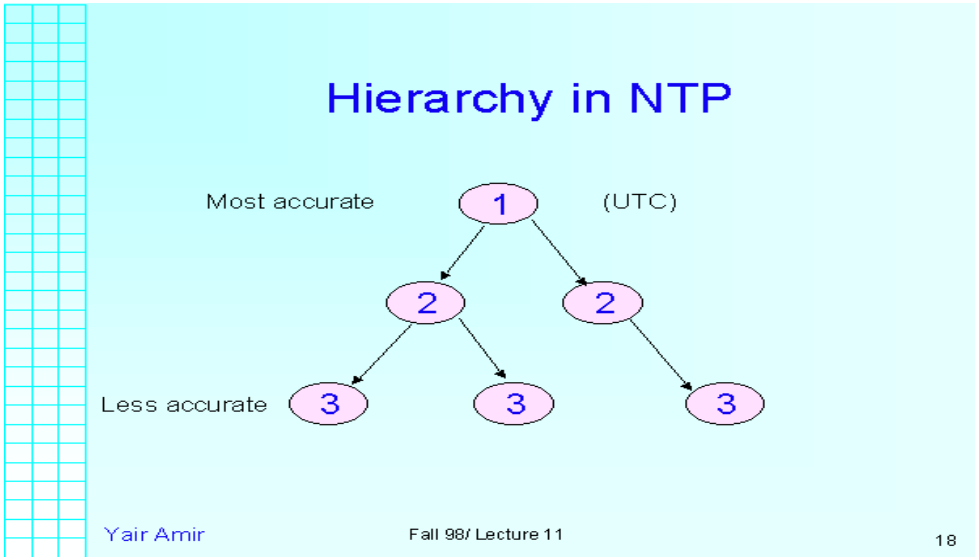
The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess an UTC server.

Decentralized Averaging Algorithm

- Each machine has a daemon without UTC
- Periodically, at fixed agreed-upon times, each machine broadcasts its local time.
- Each of them calculates the average time by averaging all the received local times.

Network Time Protocol (NTP)

- Most widely used physical clock synchronization protocol on the Internet
- 10-20 million NTP servers and clients in the Internet
- Claimed Accuracy (Varies)
  - milliseconds on WANs, submilliseconds on LANs
- Hierarchical tree of time servers.
  - The primary server at the root synchronizes with the UTC.
  - Secondary servers - backup to primary server.
  - Lowest
    - synchronization subnet with clients.



- However even with NTP, physical time cannot be perfectly synchronised. Moreover different nodes might have different NTP servers they are talking to as well.

  But, do we really need to know the physical time or do we only need to know the sequence of events and that too among dependent events?

Modeling Distributed Systems

- Can think of a distributed system as a collection of processors, and
- A communication network.
- The processors
  - May fail
  - Share no global memory
- We also assume that the network
  - Has delays that are finite but unpredictable.
  - May not respect message ordering
  - Can lose messages, duplicate messages
  - May have links that become unavailable/may fail.

A Distributed Program

distributed program is

- A set of n processes, P_1, P_2, …, P_n, typically each running on a different processor.
- A set of channels C_{ij} that connects Processor i to Processor j.

- The state of a process P_i includes
  - The local memory of P_i
  - Also depends on messages sent,

  The state of a channel C_{ij} includes
  - The messages in transit on this channel.

Types of events

- We will assume that processor sending a message does not wait for its delivery.
- Any distributed program has three types of events
  - Local actions
  - Message Send
  - Message Receive
- Can then view a distributed program as a sequential execution of the above events.

A Distributed Program in Execution

- Events can change the state of one or more process as follows.
  - A local action, aka an internal event, changes the state of the process where the event occurs
  - A send event, denoted send(m) for message m, changes the state of the process sending the message and the state of the channel that is carrying the message m.
  - A receive event, denoted recv(m) for message m, changes the state of the process that receives the message m and the state of the channel on which the message is received.

Logical Time

- Logical time is a notion we introduce to order events. It 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".

Causality between events

- Think of events happening at a process P_i.
- One can order these events sequentially,
- In other words, place a linear order on these events.
- Let $h_i = \{e^1_i, e^2_i, \ldots\}$, be the events at process P_i.
- A linear order H_i is a binary relation $\rightarrow_i$ on the events h_i such that $e^k_i \rightarrow_i e^j_i$ if and only if the event e^k_i occurs before the event e^j_i at Process Pi.
- The dependencies captured by $\rightarrow_i$ are often called as causal dependencies among the events h_i at P_i.

Causality for msgs

- What about events generated due to messages?
- Consider a binary relation $\rightarrow_{msg}$ across messages exchanged.
- Clearly, for any message m, send(m) $\rightarrow_{msg}$ recv(m).
- These two relations $\rightarrow_i$ and $\rightarrow_{msg}$ allow us to view the execution of a distributed program

Causality Definition

- One can now view the execution of a distributed program as a collection of events.
- Consider the set of events $H = \bigcup_i h_i$, where h_i is the events that occurred at process P_i.
- Define a binary relation $\rightarrow$ expressing causality among pairs of events that possibly occur at different processes. $\rightarrow$ defined as:

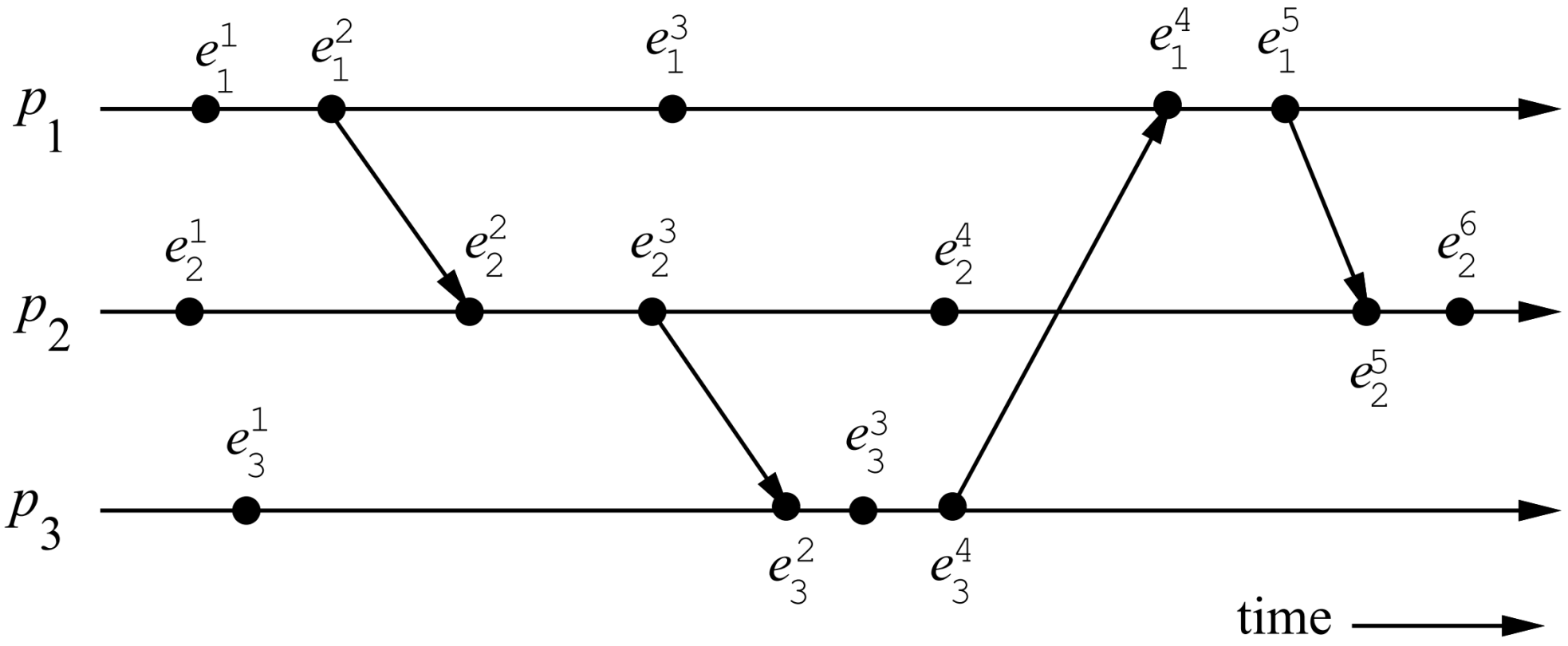
$$e^x_i \rightarrow e^y_j \text{ iff } \begin{cases} i = j \text{ and } x < y, & \text{or} \\ e^x_i \rightarrow_{msg} e^y_j, & \text{or} \\ \text{There exists } e^z_k \text{ in } H \text{ s.t. } e^x_i \rightarrow e^z_k \text{ and } e^z_k \rightarrow e^y_j \end{cases}$$

Logical vs. Physical Concurrency

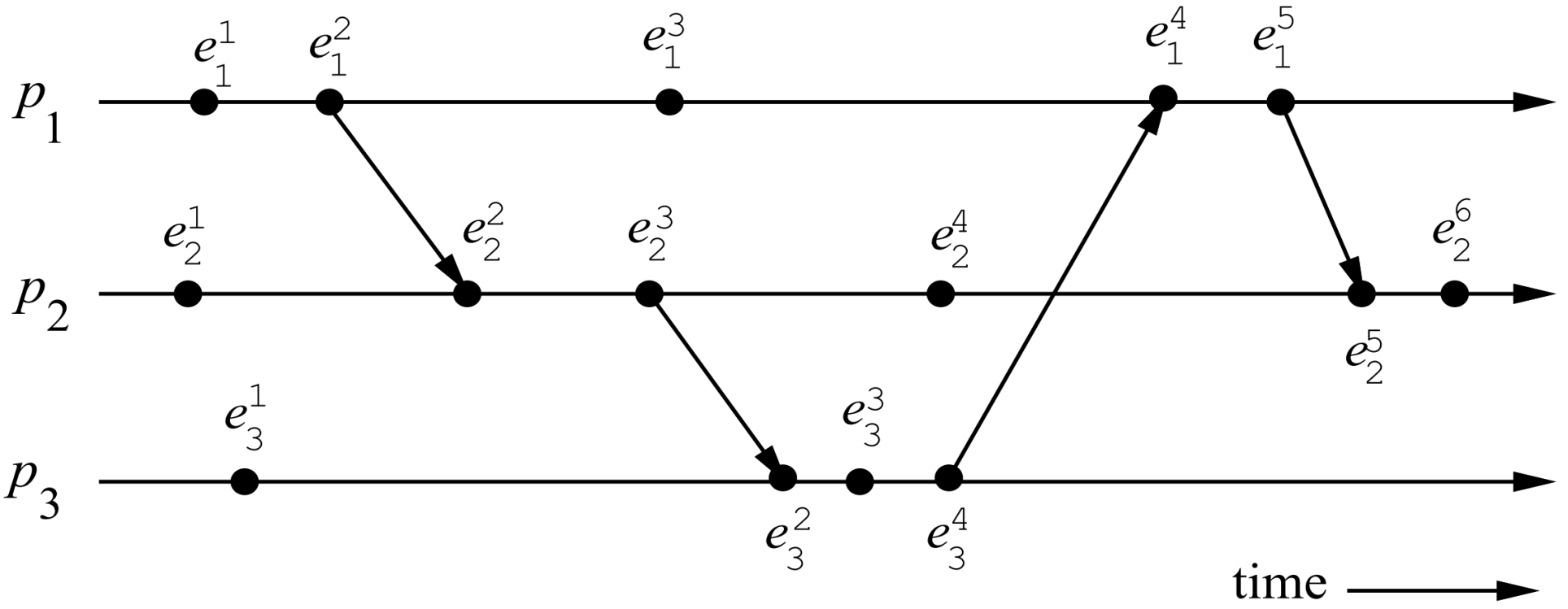
- Two events are logically concurrent if and only if the events do not causally affect each other. In other words, $e_i \parallel e_j \leftrightarrow \text{Not}(e_i \rightarrow e_j)$ and $\text{Not}(e_j \rightarrow e_i)$.
- Note that for logical concurrency of two events, the events may not occur at the same time.
- On the other hand, we define that two events are physically concurrent iff the events occur at the same physical time.

A Distributed Program in Execution

- Examples – Find concurrent events, Find some non-trivial (across P_is) precedence among events.



A Distributed Program in Execution



- Concurrent: e^1_3 and e^3_1 ? e^4_2 and e^1_3 ?
- Causal: $e^3_3 \rightarrow e^5_1$, $e^2_1 \rightarrow e^3_2$, $e^4_3 \rightarrow e^5_1$?