



IMP Note to Self



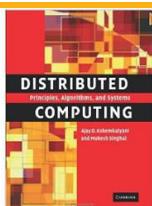
START RECORDING

Contact Session – 1

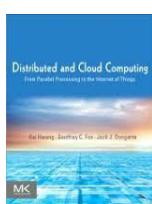
M1 - Introduction to Distributed Computing

References : T1 (Chap.1)

Text and References



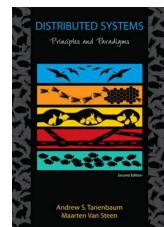
T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).



R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.



R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall

Objective of Module – 1

Introduction to Distributed Computing

- This module will give an introduction to Distributed computing models including hardware and software.
- It will also cover relevant communication / messaging models.
- Design issues and challenges for building distributed computing solutions will also be discussed.

References : T1 (Chap.1)

Contact Session – 1

M1: Introduction to Distributed Computing

- Motivation, Multiprocessor Vs Multicomputer Systems.
- Distributed Communication models: Remote Procedure Call, Publish/Subscribe model, Message Queues etc.
- Design issues and Challenges for building distributed computing systems.

Presentation Overview

- Introduction to Distributed Computing
- Motivation
- Classification and introduction to Parallel Systems
- Distributed Communication Models
- Design issues and challenges
- Summary

Topics for today

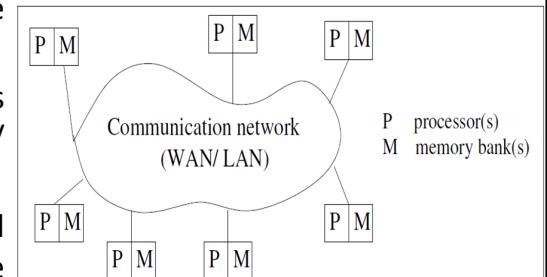
- Introduction to Distributed Computing
- Motivation
- Classification and introduction to Parallel Systems
- Distributed Communication Models
- Design issues and challenges
- Summary

DISTRIBUTED COMPUTING

- **Distributed computing** is a field of computer science that studies distributed systems.
- A distributed system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages.
- A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.

DISTRIBUTED COMPUTING

- Multiple independent machines with no physical common clock
- No shared memory but somehow a global state needs to get created when required
- May be physically distant machines and needs good network infrastructure and protocols / control algorithms to communicate
- Machines communicate via messages and somehow the system figures out the message order
- The machines doing pieces of the work can jointly figure out when a task terminates



DISTRIBUTED SYSTEM CHARACTERISTICS

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

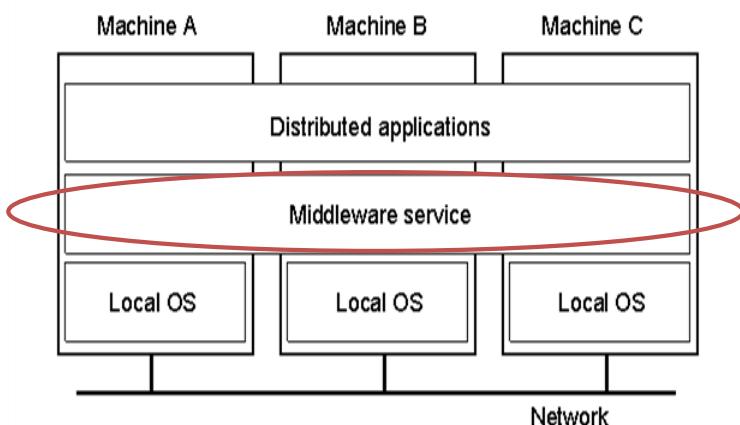
- No common physical clock
- No shared memory
- Geographical separation
- Autonomy and heterogeneity
(Different processors with different speed and OS)

RELATION TO COMPUTER SYSTEM COMPONENTS

- The distributed software is also termed as **middleware**.
- A **distributed execution** is the execution of processes across the distributed system to collaboratively achieve a common goal.
- An execution is also sometimes termed a computation or a run.
- The machines can access and use shared resources without conflicts, e.g. writing a file on a network storage
- If the program is stuck with deadlock, it can be detected and resolved.
- Independent machines can have joint decisions made via agreement protocol - e.g. majority voting by just passing messages
- All this logic can then be built into the distributed system using various architectural patterns - client/server, peer to peer etc.

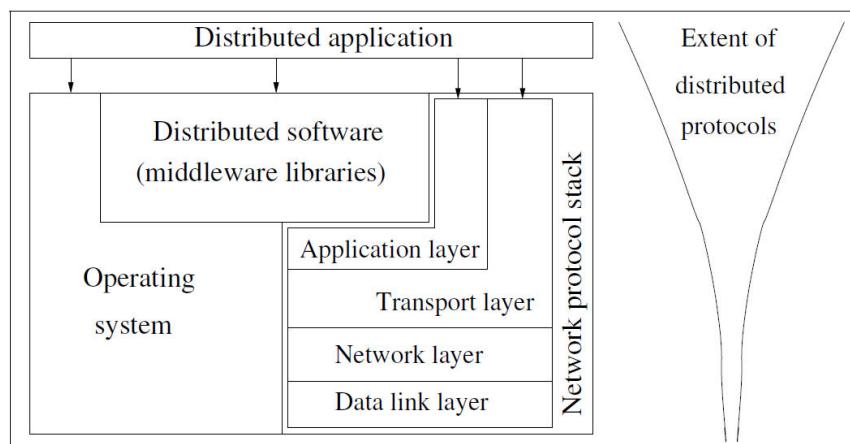
RELATION TO COMPUTER SYSTEM COMPONENTS

A Middleware Service for Distributed Applications



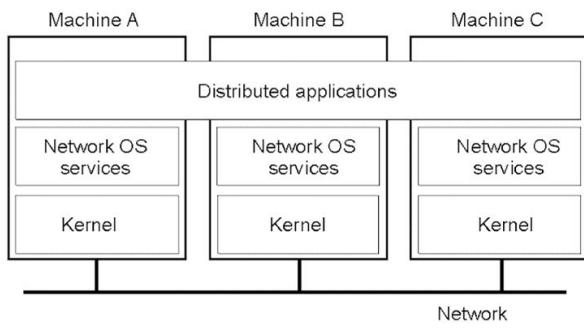
RELATION TO COMPUTER SYSTEM COMPONENTS

Relation between software components on a machine



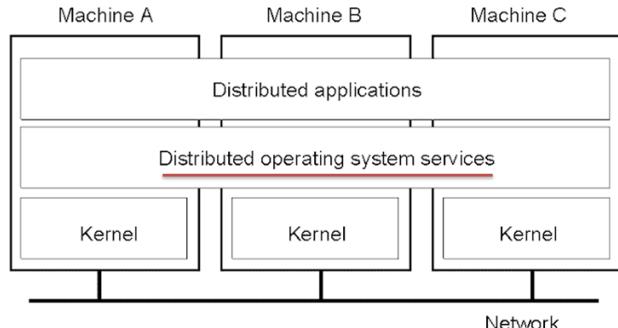
RELATION TO COMPUTER SYSTEM COMPONENTS

Complex application directly on network stack has to solve distributed computing problems



Networking OS

Easier programming where complexity is handled in distributed middleware / os services



Distributed OS

Examples of Distributed Systems

- Cluster computing systems
- Banking applications
- Internet caching systems, e.g. Akamai
- Distributed Databases
- Peer-to-peer systems for content sharing
- Media streaming systems
- Real-time process control, e.g. aircraft control systems
- IoT or sensor networks

Topics for today

- Introduction to Distributed Computing
- **Motivation**
- Classification and introduction to Parallel Systems
- Distributed Communication Models
- Design issues and challenges
- Summary

MOTIVATION

The motivation for using a distributed system is some or all of the following requirements:

1. Inherently distributed computations

- In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.

MOTIVATION

2. Resource sharing

- Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective.
- Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck. Therefore, such resources are typically distributed across the system.
- For example, distributed databases such as DB2 partition the data sets across several servers, in addition to replicating them at a few sites for rapid access as well as reliability.

MOTIVATION

3. Access to geographically remote data and resources

- In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated.
- For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site. It is therefore stored at a central server which can be queried by branch offices.
- Similarly, special resources such as supercomputers exist only in certain locations.
- Advances in the design of resource-constrained mobile devices as well as in the wireless technology with which these devices communicate have given further impetus to the importance of distributed protocols and middleware.

MOTIVATION

4. Enhanced reliability

- A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash or malfunction at the same time under normal circumstances.

Reliability entails several aspects:

1. Availability, i.e., the resource should be accessible at all times
2. Integrity, i.e., the value/state of the resource should be correct
3. Fault-tolerance, i.e., the ability to recover from system failures

MOTIVATION

5. Increased performance/cost ratio

- By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased. Although higher throughput has not necessarily been the main objective behind using a distributed system, nevertheless, any task can be partitioned across the various computers in the distributed system.
- Such a configuration provides a better performance/cost ratio than using special parallel machines. This is particularly true of the NOW configuration.

MOTIVATION

6. Scalability

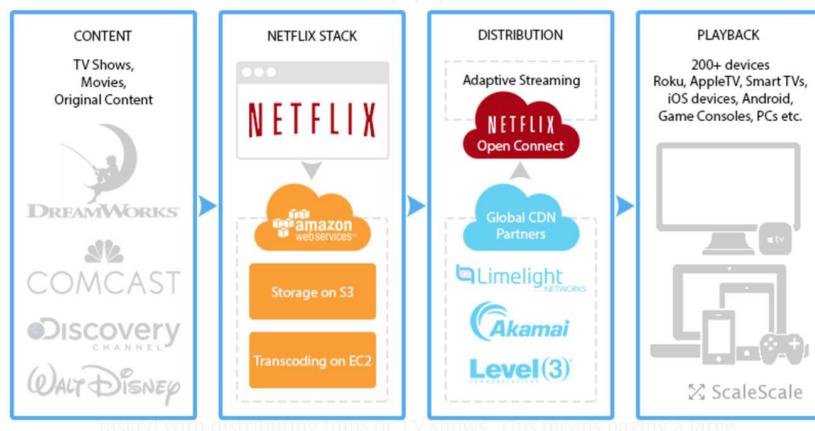
- As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network.

7. Modularity and incremental expandability

- Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms. Similarly, existing processors may be easily replaced by other processors.

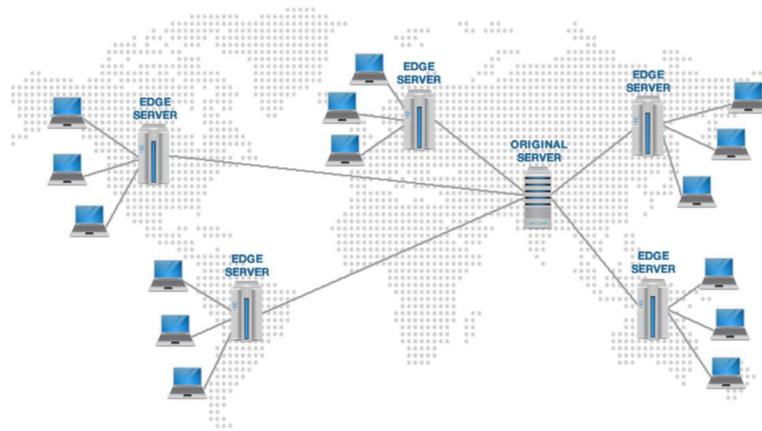
Example Distributed System : Netflix

~700+ distributed micro-services and hardware, integrated with other vendors



Example Distributed System : Netflix

Widely distributed network of content caching servers



This would be a P2P network if you were using bit torrent for free

Topics for today

- Introduction to Distributed Computing
- Motivation
- Classification and introduction to Parallel Systems
- Distributed Communication Models
- Design issues and challenges
- Summary

PARALLEL SYSTEMS

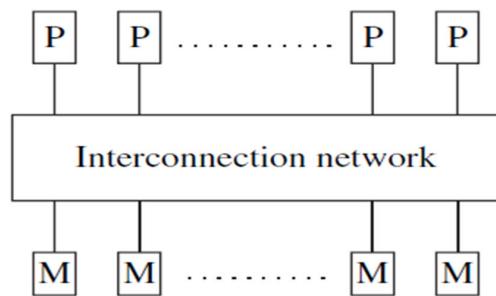
A parallel system may be broadly classified as belonging to one of three types:

1. A multiprocessor system
2. A multicompiler parallel system
3. Array processors

MULTIPROCESSOR SYSTEMS

- A multiprocessor system is a parallel system in which the multiple processors have direct access to shared memory which forms a common address space.
- Such processors usually do not have a common clock.
- A multiprocessor system usually corresponds to a uniform memory access (UMA) architecture in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same.
- The processors are in very close physical proximity and are connected by an interconnection network.
- All the processors usually run the same operating system, and both the hardware and software are very tightly coupled.

MULTIPROCESSOR SYSTEMS



Uniform memory access (UMA) multiprocessor system.

MULTIPROCESSOR SYSTEMS

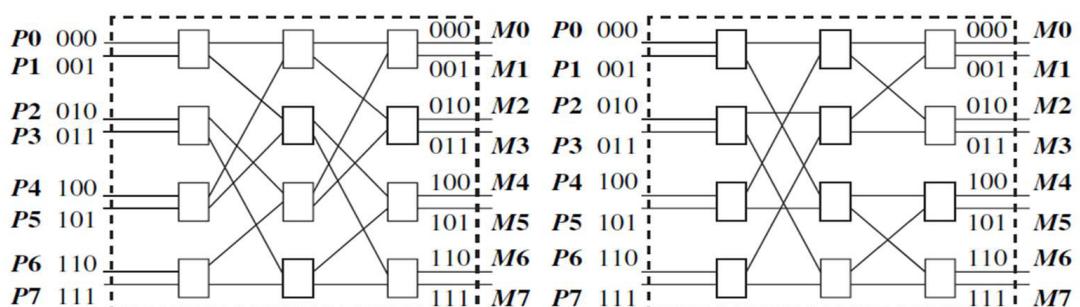
- All the processors usually run the same operating system, and both the hardware and software are very tightly coupled.
- The processors are usually of the same type, and are housed within the same box/container with a shared memory.
- The interconnection network to access the memory may be a bus, although for greater efficiency, it is usually a multistage switch with a symmetric and regular design.
- There are two popular interconnection networks
 1. The Omega network and
 2. The Butterfly network

MULTIPROCESSOR SYSTEMS

- Two popular interconnection networks – the Omega network and the Butterfly network , each of which is a multi-stage network formed of 2×2 switching elements.
- Each 2×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire.
- In a single step, however, only one data unit can be sent on an output wire. So if the data from both the input wires is to be routed to the same output wire in a single step, there is a collision.
- Various techniques such as buffering or more elaborate interconnection designs can address collisions.

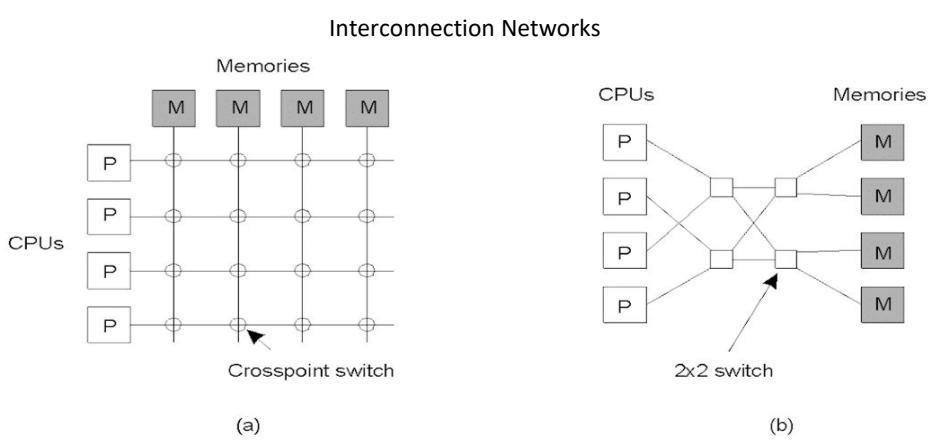
MULTIPROCESSOR SYSTEMS

INTERCONNECTION NETWORKS FOR SHARED MEMORY MULTIPROCESSOR SYSTEMS



Omega network and Butterfly network
for $n = 8$ processors P_0 – P_7 and memory banks M_0 – M_7

MULTIPROCESSOR SYSTEMS

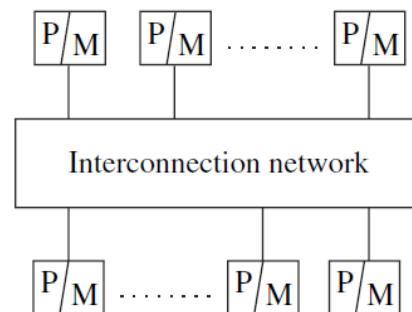


- a) A crossbar switch - faster
- b) An omega switching network - cheaper

MULTICOMPUTER PARALLEL SYSTEM

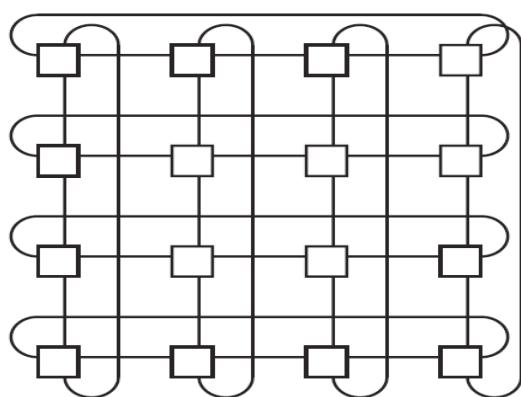
- A multicompiler parallel system is a parallel system in which the multiple processors do not have direct access to shared memory.
- The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.
- The processors are in close physical proximity and are usually very tightly coupled (homogenous hardware and software), and connected by an interconnection network.
- The processors communicate either via a common address space or via message-passing usually corresponds to a non-uniform memory access (NUMA) architecture in which the latency to access various shared memory locations from the different processors varies.

MULTICOMPUTER PARALLEL SYSTEM



Non-uniform memory access (NUMA) multiprocessor

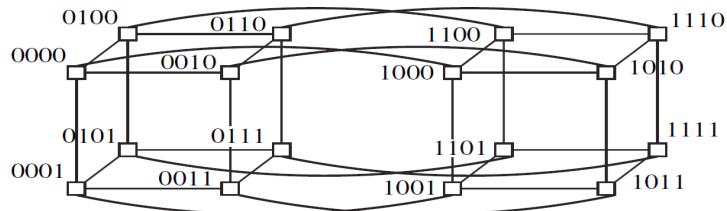
MULTICOMPUTER PARALLEL SYSTEM



A wrap-around 4×4 mesh.
For a $k \times k$ mesh which will contain k^2 processors, the maximum path length between any two processors is $2(k/2-1)$.

MULTICOMPUTER PARALLEL SYSTEM

- A k-dimensional hypercube has 2^k processor-and-memory units. Each such unit is a node in the hypercube, and has a unique k-bit label. Each of the k dimensions is associated with a bit position in the label

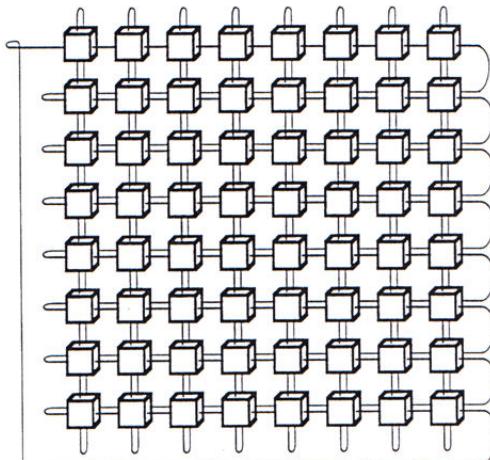


Four-dimensional hypercube.

ARRAY PROCESSORS

- Array processors belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock (but may not share memory and communicate by passing data using messages).
- Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing belong to this category.
- These applications usually involve a large number of iterations on the data.

ARRAY PROCESSORS



Flynn's Taxonomy

		Instruction Streams	
		Single	Multiple
Data Streams	Single	SISD Uniprocessors Pipelining	MISD Uncommon Fault tolerance
	Multiple	SIMD Scientific computing Matrix manipulations	MIMD Multi-computers Distributed Systems

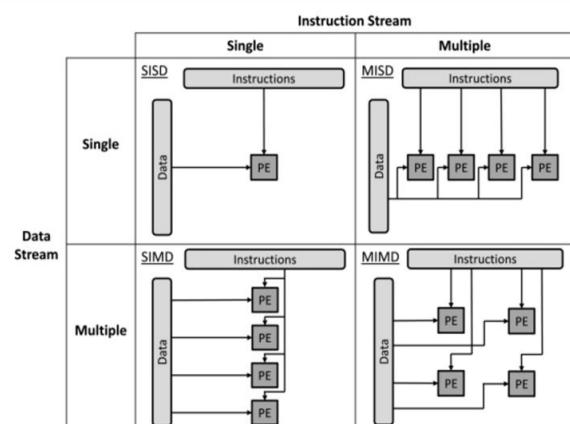


Image from sciedirect.com

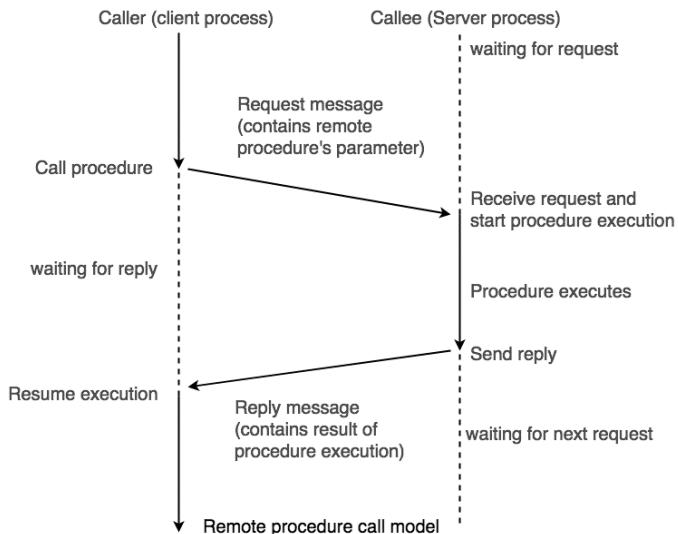
Topics for today

- Introduction to Distributed Computing
- Motivation
- Classification and introduction to Parallel Systems
- **Distributed Communication Models**
- Design issues and challenges
- Summary

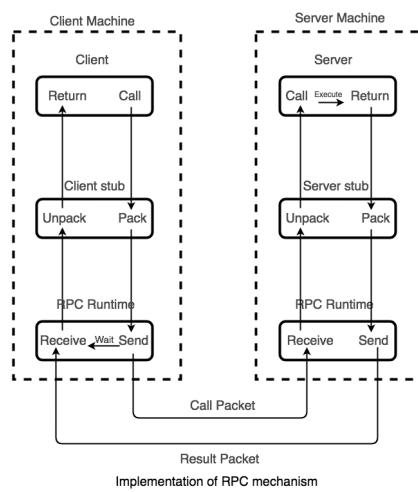
REMOTE PROCEDURE CALL (RPC)

- Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications.
- It is based on extending the conventional local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure.
- The two processes may be on the same system, or they may be on different systems with a network connecting them.

REMOTE PROCEDURE CALL (RPC)



REMOTE PROCEDURE CALL (RPC)



MESSAGE QUEUEING

- Message queueing is a method by which process (or program instances) can exchange or pass data using an interface to a system-managed queue of messages.
- Messages can vary in length and be assigned different types or usages.
- A message queue can be created by one process and used by multiple processes that read and/or write messages to the queue.
- For example, a server process can read and write messages from and to a message queue created for client processes.
- The message type can be used to associate a message with a particular client process even though all messages are on the same queue.

PUBLISH–SUBSCRIBE PATTERN

- Publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be.
- Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

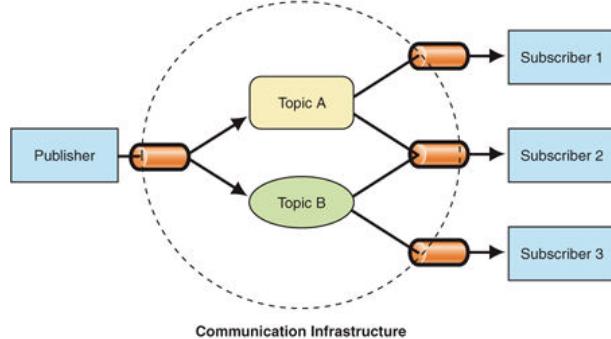
PUBLISH–SUBSCRIBE PATTERN

- Publish–subscribe is a sibling of the message queue paradigm, and is typically one part of a larger message-oriented middleware system.
- Most messaging systems support both the pub/sub and message queue models in their API, e.g. Java Message Service (JMS).
- This pattern provides greater network scalability and a more dynamic network topology, with a resulting decreased flexibility to modify the publisher and the structure of the published data.

PUBLISH–SUBSCRIBE PATTERN

- The sender (also called a publisher) uses a topic-based approach to publish messages to topic A and to topic B.
- Three receivers (also called subscribers) subscribe to these topics;
 - one receiver subscribes to topic A,
 - one receiver subscribes to topic B, and
 - one receiver subscribes to both topic A and to topic B.

PUBLISH–SUBSCRIBE PATTERN



Topics for today

- Introduction to Distributed Computing
- Motivation
- Classification and introduction to Parallel Systems
- Distributed Communication Models
- **Design issues and challenges**
- Summary

DESIGN ISSUES AND CHALLENGES

The following functions must be addressed when designing and building a distributed system:

Communication

- This task involves designing appropriate mechanisms for communication among the processes in the network. Some example mechanisms are: remote procedure call (RPC), remote object invocation (ROI), message-oriented communication versus stream-oriented communication.

Processes

- Some of the issues involved are: management of processes and threads at clients/servers; code migration; and the design of software and mobile agents.

DESIGN ISSUES AND CHALLENGES

Naming

- Naming Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a transparent and scalable manner.

Synchronization

- Synchronization Mechanisms for synchronization or coordination among the processes are essential.

Data storage and access

- Data storage and access Schemes for data storage, and implicitly for accessing the data in a fast and scalable manner across the network are important for efficiency.

DESIGN ISSUES AND CHALLENGES

Consistency and replication

- To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable

Fault tolerance

- Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes

Security

- Distributed systems security involves various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.

DESIGN ISSUES AND CHALLENGES

Applications Programming Interface (API) and transparency

- The API for communication and other specialized services is important for the ease of use and wider adoption of the distributed systems services by non-technical users.

Scalability and modularity

- The algorithms, data (objects), and services must be as distributed as possible. Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

Topics for today

- Introduction to Distributed Computing
- Motivation
- Classification and introduction to Parallel Systems
- Distributed Communication Models
- Design issues and challenges
- **Summary**

SUMMARY

- Basic concepts and motivation of a Parallel and Distributed Systems
- Challenges in systems, algorithms, new applications

Next Session

Logical Time :

- How do we logically correlate time across distributed machines with no common clock

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

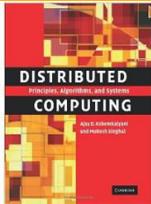
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



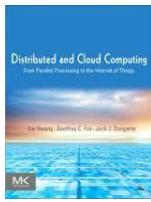
CS-2 : Logical Clocks

[T1: Chap - 3]

Text and References



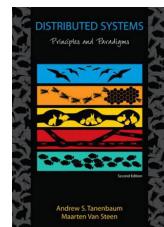
T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).



R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.



R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall

Objective of Module – 2

Logical Clocks & Vector Clocks

- This module will explore ways of proving and managing a global notion of time (or clock) in a distributed computing system where there is no centralized hardware clock is available or feasible.

Contact Session – 2

M2: Logical Clocks & Vector clocks

- A framework for a system of logical clocks.
- Scalar time, Vector time.
- Implementation of Logical and Vector clocks, Efficient implementation of Vector clocks.
- Physical Clock synchronization: NTP

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Presentation Overview

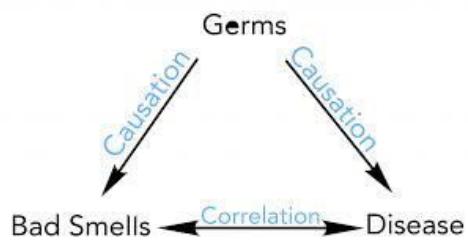
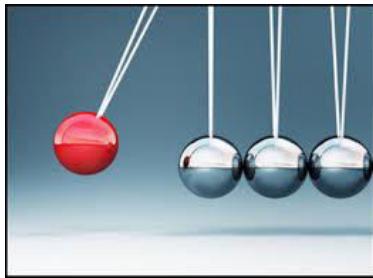
- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Logical time introduction

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time.
- In distributed systems, it is not possible to have global physical time; it is possible to realize only an approximation of it.
- Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.

Logical time introduction

- Causality is influence by which one event, process or state contributes to the production of another event, process or state where the cause is partly responsible for the effect, and the effect is partly dependent on the cause.



Logical time introduction

The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems some of these problems is as follows.

Distributed algorithms design

The knowledge of the causal precedence relation among events helps ensure liveness and fairness in mutual exclusion algorithms, helps maintain consistency in replicated databases, and helps design correct deadlock detection algorithms to avoid phantom and undetected deadlocks.

Logical time introduction

Tracking of dependent events

In distributed debugging, the knowledge of the causal dependency among events 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.

Knowledge about the progress

The knowledge of the causal dependency among events helps measure the progress of processes in the distributed computation. This is useful in discarding obsolete information, garbage collection, and termination detection.

Logical time introduction

Concurrency measure

The knowledge of how many events are causally dependent is useful in measuring the amount of concurrency in a computation. All events that are not causally related can be executed concurrently. Thus, an analysis of the causality in a computation gives an idea of the concurrency in the program.

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Implementing logical clocks

Implementation of logical clocks requires addressing two issues

- ✓ Data structures local to every process to represent logical time and
- ✓ A protocol (set of rules) to update the data structures to ensure the consistency condition.

Implementing logical clocks

Each process p_i maintains data structures that allow it the following two capabilities:

- A **local logical clock**, denoted by lc_i , that helps process p_i measure its own progress.
- A **logical global clock**, denoted by gc_i , that is a representation of process p_i 's local view of the logical global time. It allows this process to assign consistent timestamps to its local events. Typically, lc_i is a part of gc_i .

Implementing logical clocks

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

R1 : This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).

R2 : This rule governs how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Scalar time

Time domain in this representation is the set of non-negative integers. The logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i . Rules R1 and R2 to update the clocks are as follows:

R1 Before executing an event (send, receive, or internal), process p_i executes the following:

$$C_i = C_i + d \quad (d > 0)$$

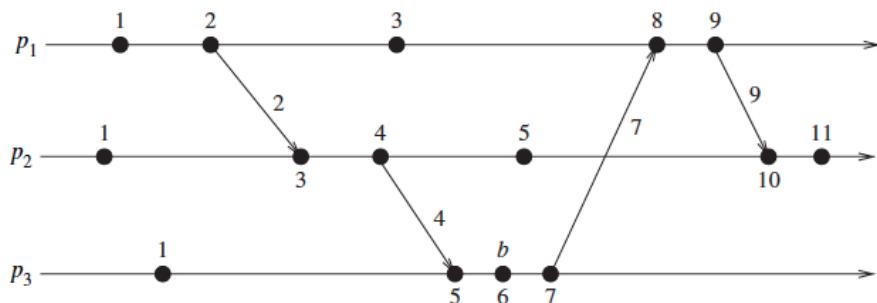
Every time R1 is executed, d can have a different value, and this value may be application-dependent typically d is kept at 1 because this is able to identify the time of each event uniquely at a process, while keeping the rate of increase of d to its lowest level.

Scalar time

R2 Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg} , it executes the following actions:

1. $C_i = \max(C_i, C_{msg})$;
2. execute R1;
3. deliver the message.

Scalar time



Evolution of scalar time with $d=1$

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Lamport timestamps

- The algorithm of **Lamport timestamps** is a simple algorithm used to determine the order of events in a distributed computer system.
- As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method.
- They are named after their creator, Leslie Lamport.

Lamport timestamps

- Distributed algorithms such as resource synchronization often depend on some method of ordering events to function.
- For example, consider a system with two processes and a disk.
- The processes send messages to each other, and also send messages to the disk requesting access.
- The disk grants access in the order the messages were sent.
- Now, imagine process 1 sends a message to the disk asking for access to write, and then sends a message to process 2 asking it to read.
- Process 2 receives the message, and as a result sends its own message to the disk.

Lamport timestamps

- Now, due to some timing delay, the disk receives both messages at the same time: how does it determine which message happened-before the other?
- A logical clock algorithm provides a mechanism to determine facts about the order of such events.

Lamport timestamps

- Lamport invented a simple mechanism by which the happened-before ordering can be captured numerically.
- A Lamport logical clock is an incrementing software counter maintained in each process. It follows some simple rules:
 1. A process increments its counter before each event in that process;
 2. When a process sends a message, it includes its counter value with the message;
 3. On receiving a message, the receiver process sets its counter to be the maximum of the message counter and its own counter incremented, before it considers the message received.

Lamport timestamps

- Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes.
- When a process receives a message, it resynchronizes its logical clock with that sender

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- **Vector Time**
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Vector time

- Each process p_i maintains a vector $vt_i(1...n)$, where $vt_i(i)$ is the local logical clock of p_i and describes the logical time progress at process p_i .
- $Vt_i(j)$ represents process p_i 's latest knowledge of process p_j local time.
- If $vt_i(j) = x$, then process p_i knows that local time at process p_j has progressed till x .
- The entire vector vt_i constitutes p_i 's view of the global logical time and is used to timestamp events.

Vector time

Process p_i uses the following two rules R1 and R2 to update its clock:

R1 Before executing an event, process p_i updates its local logical time as follows:

$$vt_i(i) = vt_i(i) + d \quad d > 0$$

Vector time

R2 Each message m is piggybacked with the vector clock v_t of the sender process at sending time. On the receipt of such a message (m, v_t) , process p_i executes the following sequence of actions:

1. update its global logical time as

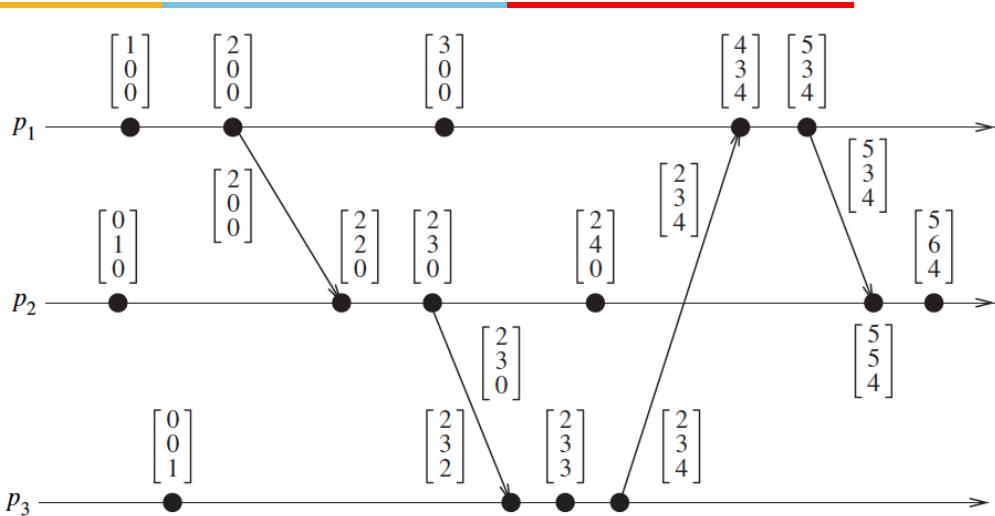
$$1 \leq k \leq n : vt_i(k) = \max(vt_i(k), vt(k))$$

2. execute R1 ;

3. deliver the message m .

The timestamp associated with an event is the value of the vector clock of its process when the event is executed.

Vector time



Vector time

- **Vector clocks** is an algorithm for generating a partial ordering of events in a distributed system and detecting causality violations.
- Just as in Lamport timestamps, interprocess messages contain the state of the sending process's logical clock.
- A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process; a local "smallest possible values" copy of the global clock-array is kept in each process, with the following rules for clock updates:

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal-kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Rules for vector clocks updates

- Initially all clocks are zero.
- Each time a process experiences an internal event, it increments its own logical clock in the vector by one.
- Each time a process prepares to send a message, it sends its entire vector along with the message being sent.
- Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element).

Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal-Kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

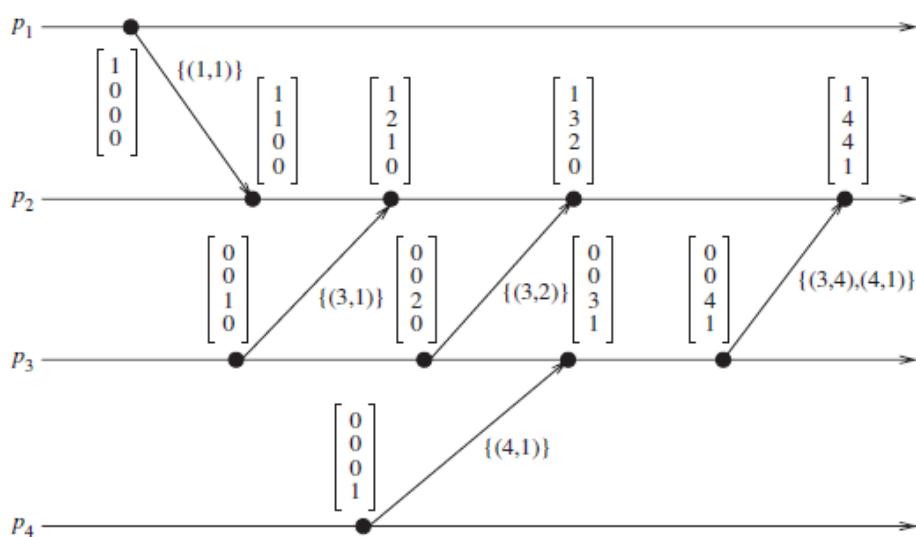
Singhal–Kshemkalyani's differential technique

- *Singhal–Kshemkalyani's differential technique* is based on the observation that between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change.
- This is more likely when the number of processes is large because only a few of them will interact frequently by passing messages.
- In this technique, when a process p_i sends a message to a process p_j , it piggybacks only those entries of its vector clock that differ since the last message sent to p_j .

Singhal-Kshemkalyani's differential technique

- Thus this technique cuts down the message size, communication bandwidth and buffer (to store messages) requirements.
- In the worst of case, every element of the vector clock has been updated at p_i since the last message to process p_j , and the next message from p_i to p_j will need to carry the entire vector timestamp of size n . However, on the average the size of the timestamp on a message will be less than n .

Singhal-Kshemkalyani's differential technique



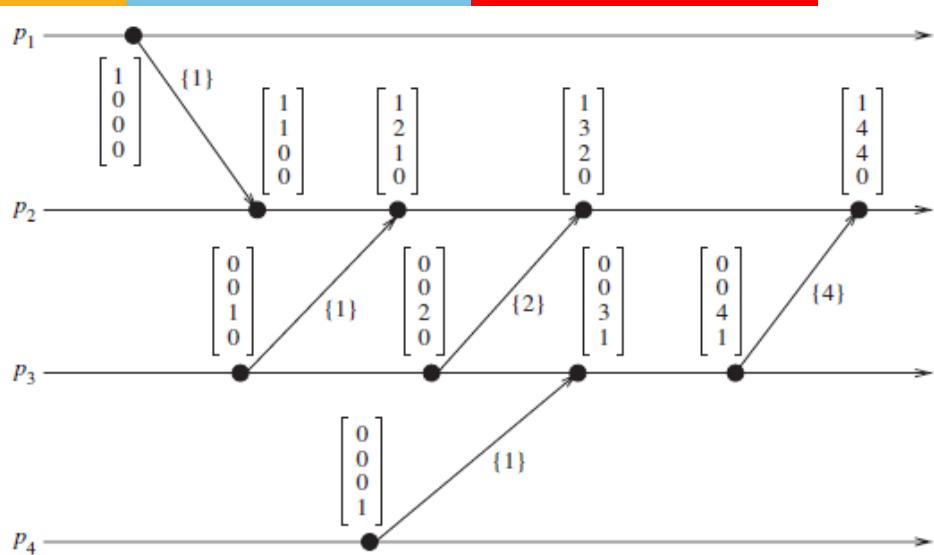
Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Fowler–Zwaenepoel's direct-dependency technique

- Fowler–Zwaenepoel direct dependency technique reduces the size of messages by transmitting only a scalar value in the messages.
- No vector clocks are maintained on-the-fly.
- Instead, a process only maintains information regarding direct dependencies on other processes.
- A vector time for an event, which represents transitive dependencies on other processes, is constructed off-line from a recursive search of the direct dependency information at processes.

Fowler–Zwaenepoel's direct-dependency technique



Presentation Overview

- Logical Time Introduction
- Implementing Logical Clocks
- Scalar Time
- Lamport Timestamps
- Vector Time
- Rules For Vector Clocks Updates
- Singhal–kshemkalyani's Differential Technique
- Fowler – Zwaenepoel's Direct-dependency Technique
- Physical Clock Synchronization: NTP

Physical clock synchronization: NTP

- In centralized systems, there is no need for clock synchronization because, generally, there is only a single clock.
- A process gets the time by simply issuing a system call to the kernel.
- When another process after that tries to get the time, it will get a higher time value.
- Thus, in such systems, there is a clear ordering of events and there is no ambiguity about the times at which these events occur.

Physical clock synchronization: NTP

- In distributed systems, there is no global clock or common memory.
- Each processor has its own internal clock and its own notion of time.
- In practice, these clocks can easily drift apart by several seconds per day, accumulating significant errors over time.
- Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start.
- This clearly poses serious problems to applications that depend on a synchronized notion of time

Physical clock synchronization: NTP

- Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time.
- It has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values.
- It is quite common that distributed applications and network protocols use timeouts, and their performance depends on how well physically dispersed processors are time synchronized.
- Design of such applications is simplified when clocks are synchronized.

Physical clock synchronization: NTP

- Due to different clock rates, the clocks at various sites may diverge with time, and periodically a clock synchronization must be performed to correct this clock skew in distributed systems.
- Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).
- Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed *physical clocks*.

Physical clock synchronization: NTP

- The *Network Time Protocol (NTP)* , which is widely used for clock synchronization on the Internet, uses the the *offset delay estimation* method.
- The design of NTP involves a hierarchical tree of time servers.
- The primary server at the root synchronizes with the UTC.
- The next level contains secondary servers, which act as a backup to the primary server.
- At the lowest level is the synchronization subnet which has the clients.

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

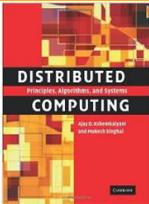
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



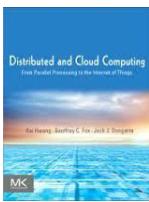
CS-3 : Global Snapshot

[T1: Chap – 4]

Text and References



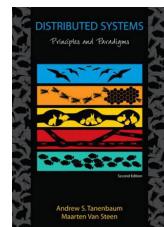
T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).



R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.



R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall

Objective of Module – 3

This module will

- (i) highlight the complexity of recording a global state in the absence of a single physical memory and a single physical clock in a distributed system and
- (ii) explore ways or algorithms to record or collect global state of a distributed computation.

Contact Session – 3

Global state and snapshot recording algorithms

- System model and definitions
- Snapshot recording algorithms for FIFO channels
- Snapshot recording algorithms for non-FIFO channels
- Necessary and sufficient conditions for consistent global snapshots.

Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

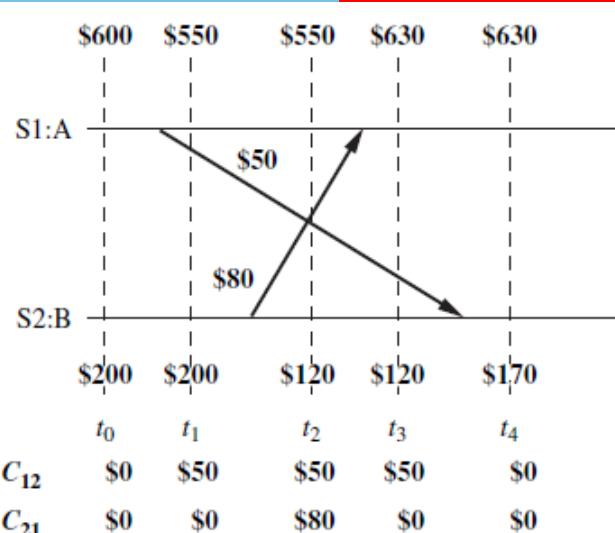
Introduction

- Recording the global state of a distributed system on-the-fly is an important paradigm when one is interested in analyzing, testing, or verifying properties associated with distributed executions.
- Unfortunately, the lack of both a globally shared memory and a global clock in a distributed system, added to the fact that message transfer delays in these systems are finite but unpredictable
- The distributed nature of the local clocks and local memory makes it difficult to record the global state of the system efficiently.
- If shared memory were available, an up-to-date state of the entire system would be available to the processes sharing the memory.

Introduction

- The global state of a distributed system is a collection of the local states of its components.
- For examples, in detection of stable properties such as deadlocks and termination, global state of the system is examined for certain properties; for failure recovery, a global state of the distributed system (called a checkpoint) is periodically saved and recovery from a processor failure is done by restoring the system to the last saved global state; for debugging distributed software, the system is restored to a consistent global state and the execution resumes from there in a controlled manner

Introduction



Introduction

- Suppose the local state of Account A is recorded at time t_0 to show \$600 and the local state of Account B and channels C12 and C21 are recorded at time t_2 to show \$120, \$50, and \$80, respectively.
- Then the recorded global state shows \$850 in the system.
- An extra \$50 appears in the system.
- The reason for the inconsistency is that Account A's state was recorded before the \$50 transfer to Account B using channel C12 was initiated, whereas channel C12's state was recorded after the \$50 transfer was initiated.

Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

System model and definitions

- The system consists of a collection of n processes, p_1, p_2, \dots, p_n , that are connected by channels.
- There is no globally shared memory and processes communicate solely by passing messages.
- There is no physical global clock in the system.
- Message send and receive is asynchronous. Messages are delivered reliably with finite but arbitrary time delay.
- The system can be described as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.

System model and definitions

- Let C_{ij} denote the channel from process p_i to process p_j .
- Processes and channels have states associated with them.
- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc., and may be highly dependent on the local context of the distributed application.
- The state of channel C_{ij} , denoted by SC_{ij} , is given by the set of messages in transit in the channel.

Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

A consistent global state

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state GS is defined as

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}.$$

A consistent global state

- A global state GS is a *consistent global state* iff it satisfies the following two conditions

C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$ (\oplus is the Ex-OR operator).

C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$.

Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- **Interpretation in terms of cuts**
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.’s algorithm
 - Mattern’s algorithm

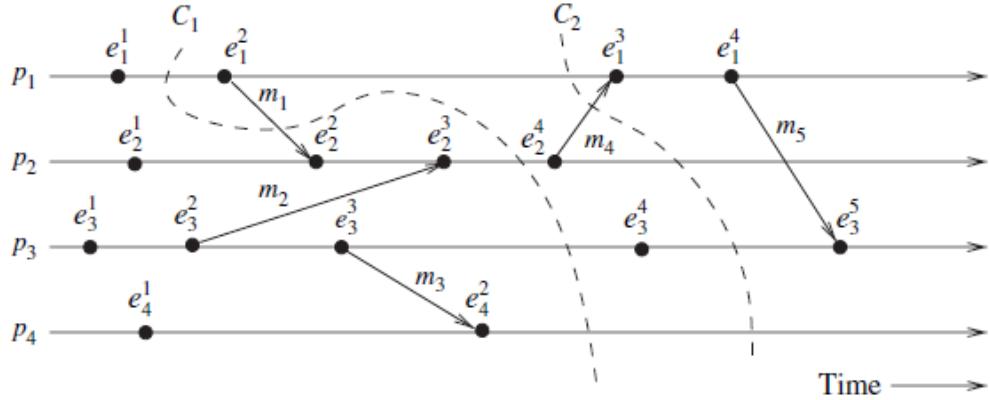
Interpretation in terms of cuts

- Cuts in a space–time diagram provide a powerful graphical aid in representing and reasoning about the global states of a computation.
- A cut is a line joining an arbitrary point on each process line that slices the space–time diagram into a PAST and a FUTURE

Interpretation in terms of cuts

- A consistent global state corresponds to a cut in which every message received in the PAST of the cut has been sent in the PAST of that cut.
- Such a cut is known as a *consistent cut*.
- All the messages that cross the cut from the PAST to the FUTURE are captured in the corresponding channel state.
- For example, consider the space–time diagram for the computation illustrated
- Cut C1 is inconsistent because message m1 is flowing from the FUTURE to the PAST.
- Cut C2 is consistent and message m4 must be captured in the state of channel C21

Interpretation in terms of cuts



Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

Issues in recording a global state

- If a global physical clock were available, the following simple procedure could be used to record a consistent global snapshot of a distributed system.
- In this, the initiator of the snapshot collection decides a future time at which the snapshot is to be taken and broadcasts this time to every process.
- All processes take their local snapshots at that instant in the global time.
- The snapshot of channel C_{ij} includes all the messages that process p_j receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot.
- All messages are timestamped with the sender's clock.

Issues in recording a global state

However, a global physical clock is not available in a distributed system and the following two issues need to be addressed in recording of a consistent global snapshot of a distributed system

I1: How to distinguish between the messages to be recorded in the snapshot (either in a channel state or a process state) from those not to be recorded.

The answer to this comes from conditions **C1** and **C2** as follows:

- Any message that is sent by a process before recording its snapshot must be recorded in the global snapshot (from **C1**).
- Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

Issues in recording a global state

I2: How to determine the instant when a process takes its snapshot.

The answer to this comes from condition **C2** as follows:

- A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

Chandy–Lamport algorithm

- The Chandy-Lamport algorithm uses a control message, called a *marker*.
- After a site has recorded its snapshot, it sends a *marker* along all of its outgoing channels before sending out any more messages.
- Since channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot (i.e., channel state or process state) from those not to be recorded in the snapshot.
- This addresses issue **I1**. The role of markers in a FIFO system is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition **C2**.

Chandy–Lamport algorithm

- A process initiates snapshot collection by executing the *marker sending rule* by which it records its local state and sends a marker on each outgoing channel.
- A process executes the *marker receiving rule* on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the *marker sending rule* to record its local state.
- Otherwise, the state of the incoming channel on which the marker is received is recorded as the set of computation messages received on that channel after recording the local state but before receiving the marker on that channel.
- The algorithm can be initiated by any process by executing the *marker sending rule*. The algorithm terminates after each process has received a marker on all of its incoming channels.

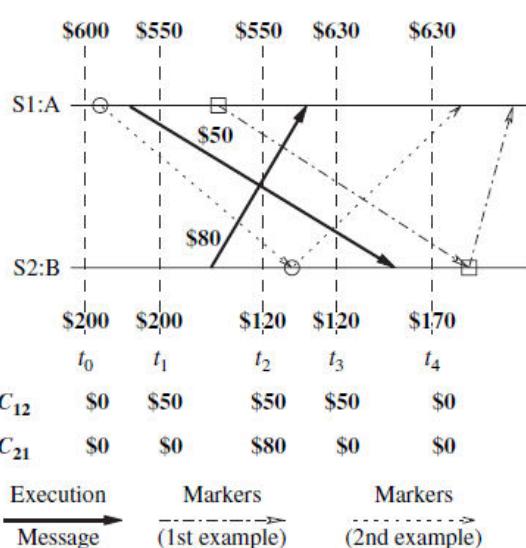
Chandy–Lamport algorithm

- The recorded local snapshots can be put together to create the global snapshot in several ways.
- One policy is to have each process send its local snapshot to the initiator of the algorithm.
- Another policy is to have each process send the information it records along all outgoing channels, and to have each process receiving such information for the first time propagate it along its outgoing channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy–Lamport algorithm

- Multiple processes can initiate the algorithm concurrently.
- If multiple processes initiate the algorithm concurrently, each initiation needs to be distinguished by using unique markers.
- Different initiations by a process are identified by a sequence number

Chandy–Lamport algorithm



Presentation Overview

- Introduction
- System model and definitions
- A consistent global state
- Interpretation in terms of cuts
- Issues in recording a global state
- Snapshot algorithms for FIFO channels
 - Chandy–Lamport algorithm
- Snapshot algorithms for non-FIFO channels
 - Non-FIFO algorithm of Helary
 - Lai–Yang algorithm
 - Li et al.'s algorithm
 - Mattern's algorithm

Snapshot algorithms for non-FIFO channels

- In a non-FIFO system, either some degree of
 - Inhibition : temporarily delaying the execution of an application
 - piggybacking : delaying the send of a computation message
- The non-FIFO algorithm by Helary uses message inhibition.
- The non-FIFO algorithms by Lai and Yang, Li er nl and Mattern use message piggybacking to distinguish computation messages sent after the marker from those sent before the marker.

Non-FIFO algorithm of Helary

The non-FIFO algorithm of Helary uses message inhibition to avoid an inconsistency in a global snapshot in the following way:

- When a process receives a marker, it immediately returns an acknowledgement
- After a process has sent a marker on the outgoing channel to process j , it does not send any messages on this channel until it is sure that j has recorded its local state.
- Process i can conclude this if it has received an acknowledgement for the marker sent to j , or has received a marker for this snapshot from j .

Lai–Yang algorithm

The Lai–Yang algorithm fulfills this role of a marker in a non-FIFO system by using a coloring scheme on computation messages that works as follows:

1. Every process is initially white and turns red while taking a snapshot. The equivalent of the “marker sending rule” is executed when a process turns red.
2. Every message sent by a white (red) process is colored white (red). Thus, a white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot.
3. Every white process takes its snapshot at its convenience, but no later than the instant it receives a red message.

Lai–Yang algorithm

4. Every white process records a history of all white messages sent or received by it along each channel.
5. When a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot.
6. The initiator process evaluates transit LS_i and LS_j to compute the state of a channel C_{ij}

Lai–Yang algorithm

- Though marker messages are not required in the algorithm, each process has to record the entire message history on each channel as part of the local snapshot.
- Thus, the space requirements of the algorithm may be large.
- However, in applications (such as termination detection) where the number of messages in transit in a channel is sufficient, message histories can be replaced by integer counters reducing the space requirement.
- Lai and Yang describe how the size of the local storage and snapshot recording can be reduced by storing only the messages sent and received since the previous snapshot recording, assuming that the previous snapshot is still available.
- This approach can be very useful in applications that require repeated snapshots of a distributed system.

Li et al.'s algorithm

- A process maintains two counters for each incident channel to record the number of messages sent and received on the channel and reports these counter values with its snapshot to the initiator.
- This simplification is combined with the incremental technique to compute channel states, which reduces the size of message histories to be stored and transmitted.
- The initiator computes the state of C_{ij} as: (the number of messages in C_{ij} in the previous snapshot) + (the number of messages sent on C_{ij} since the last snapshot at process p_i) – (the number of messages received on C_{ij} since the last snapshot at process p_j).

Li et al.'s algorithm

4. Every white process records a history, as input and output counters, of all white messages sent or received by it along each channel after the previous snapshot (by the same initiator).
5. When a process turns red, it sends these histories (i.e., input and output counters) along with its snapshot to the initiator process that collects the global snapshot.
6. The initiator process computes the state of channel C_{ij}

Mattern's algorithm

Mattern's algorithm assumes a single initiator process and works as follows:

1. The initiator "ticks" its local clock and selects a future vector time s at which it would like a global snapshot to be recorded. It then broadcasts this time s and freezes all activity until it receives all acknowledgements of the receipt of this broadcast.
2. When a process receives the broadcast, it remembers the value s and returns an acknowledgement to the initiator.
3. After having received an acknowledgement from every process, the initiator increases its vector clock to s and broadcasts a dummy message to all processes. (Observe that before broadcasting this dummy message, the local clocks of other processes have a value $\geq s$.)

Mattern's algorithm

4. The receipt of this dummy message forces each recipient to increase its clock to a value $\geq s$ if not already $\geq s$.
5. Each process takes a local snapshot and sends it to the initiator when (just before) its clock increases from a value less than s to a value $\geq s$. Observe that this may happen before the dummy message arrives at the process.
6. The state of C_{ij} is all messages sent along C_{ij} , whose timestamp is smaller than s and which are received by p_j after recording LS_j .

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

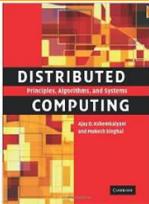
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



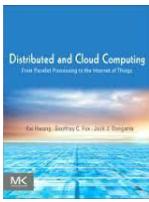
CS-4 : Distributed Algorithms – Design

[T1: Chap – 5]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

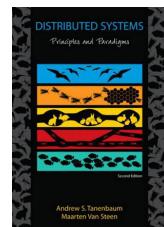


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 4

Distributed algorithms – Design

- This module will introduce a setting in which distributed algorithms can be designed, expressed, and analyzed.
- Algorithms for a few basic graph problems in a distributed environment will also be covered.

Contact Session – 4

M4: Distributed algorithms – Design

- Terminology and basic algorithms
- Classifications and basic concepts
- Elementary graph algorithms, Synchronizers.
- Maximal Independent set, and Connected dominating set.

Presentation Overview

- Classifications and basic concepts
- Elementary graph algorithms
- Synchronizers
- Maximal independent set (MIS)
- Connected dominating set

Classifications And Basic Concepts

Classifications And Basic Concepts

1. Application Executions And Control Algorithm Executions
2. Centralized And Distributed Algorithms
3. Symmetric And Asymmetric Algorithms
4. Anonymous Algorithms
5. Uniform Algorithms
6. Adaptive Algorithms
7. Deterministic Versus Non-deterministic Executions
8. Execution Inhibition
9. Synchronous And Asynchronous Systems
10. Online Versus Offline Algorithms
11. Failure Models
12. Wait-free Algorithms
13. Communication Channels

1. APPLICATION EXECUTIONS AND CONTROL ALGORITHM EXECUTIONS

In many cases, a control algorithm also needs to be executed in order to monitor the application execution or to perform various auxiliary functions such as:

- creating a spanning tree,
- creating a connected dominating set,
- achieving consensus among the nodes,
- distributed transaction commit,
- distributed deadlock detection,
- global predicate detection,
- termination detection,
- global state recording,
- check pointing and memory consistency enforcement in distributed shared memory systems.

2. CENTRALIZED AND DISTRIBUTED ALGORITHMS

- In a distributed system, a **centralized algorithm** is one in which a pre dominant amount of work is performed by one (or possibly a few) processors, whereas other processors play a relatively smaller role in accomplishing the joint task.
- The roles of the other processors are usually confined to requesting information or supplying information, either periodically or when queried.
- A typical system configuration suited for centralized algorithms is the client–server configuration.

2. CENTRALIZED AND DISTRIBUTED ALGORITHMS

- From a theoretical perspective, the single server is a potential bottleneck for both processing and bandwidth access on the links.
- The single server is also a single point of failure.
- Of course, these problems are alleviated in practice by using replicated servers distributed across the system, and then the overall configuration is not as centralized any more.

2. CENTRALIZED AND DISTRIBUTED ALGORITHMS

- A **distributed algorithm** is one in which each processor plays an equal role in sharing the message overhead, time overhead, and space overhead.
- It is difficult to design a purely distributed algorithm (that is also efficient) for some applications.
- Consider the problem of recording a global state of all the nodes.
- The well-known Chandy–Lamport algorithm is distributed yet one node, which is typically the initiator, is responsible for assembling the local states of the other nodes, and hence plays a slightly different role.

3. SYMMETRIC AND ASYMMETRIC ALGORITHMS

- A **symmetric algorithm** is an algorithm in which all the processors execute the same logical functions.
- An **asymmetric algorithm** is an algorithm in which different processors execute logically different functions.
- A centralized algorithm is always asymmetric.
- An algorithm that is not fully distributed is also asymmetric.
- In the client–server configuration, the clients and the server execute asymmetric algorithms.

4. ANONYMOUS ALGORITHMS

- An **anonymous system** is a system in which neither processes nor processors use their process identifiers and processor identifiers to make any execution decisions in the distributed algorithm.
- An anonymous algorithm is an algorithm which runs on an anonymous system and therefore does not use process identifiers or processor identifiers in the code.

5. UNIFORM ALGORITHMS

- A **uniform algorithm** is an algorithm that does not use n , the number of processes in the system, as a parameter in its code.
- A uniform algorithm is desirable because it allows scalability transparency, and processes can join or leave the distributed execution without intruding on the other processes, except its immediate neighbors that need to be aware of any changes in their immediate topology

6. ADAPTIVE ALGORITHMS

- Consider the context of a problem X.
- In a system with n nodes, let k ($k \leq n$) be the number of nodes “participating” in the context of X when the algorithm to solve X is executed.
- If the complexity of the algorithm can be expressed in terms of k rather than in terms of n, the algorithm is **adaptive**.

7. DETERMINISTIC VERSUS NON-DETERMINISTIC EXECUTIONS

- A **deterministic** receive primitive specifies the source from which it wants to receive a message.
- A **non-deterministic** receive primitive can receive a message from any source.

8. EXECUTION INHIBITION

- Protocols that require processors to suspend their normal execution until some series of actions stipulated by the protocol have been performed are termed as **inhibitory or freezing protocols**

9. SYNCHRONOUS AND ASYNCHRONOUS SYSTEMS

- A *synchronous system* is a system that satisfies the following properties:
 - There is a known upper bound on the message communication delay.
 - There is a known bounded drift rate for the local clock of each processor with respect to real-time. The drift rate between two clocks is defined as the rate at which their values diverge.
 - There is a known upper bound on the time taken by a process to execute a logical step in the execution.
- An *asynchronous system* is a system in which none of the above three properties of synchronous systems are satisfied.

10. ONLINE VERSUS OFFLINE ALGORITHMS

- An **on-line algorithm** is an algorithm that executes as the data is being generated.
- An **off-line algorithm** is an algorithm that requires all the data to be available before algorithm execution begins.
- Clearly, on-line algorithms are more desirable.
- Debugging and scheduling are two example areas where online algorithms offer clear advantages.

11. FAILURE MODELS

Fail-stop

- In this model, a properly functioning process may fail by stopping execution from some instant thenceforth.
- Additionally, other processes can learn that the process has failed.

Crash

- In this model, a properly functioning process may fail by stopping to function from any instance thenceforth.

Receive omission

- A properly functioning process may fail by intermittently receiving only some of the messages sent to it, or by crashing.

11. FAILURE MODELS

Send omission

- A properly functioning process may fail by intermittently sending only some of the messages it is supposed to send, or by crashing.

General omission

- A properly functioning process may fail by exhibiting either or both of send omission and receive omission failures.

11. FAILURE MODELS

Byzantine or malicious failure, with authentication

- In this model, a process may exhibit any arbitrary behavior. However, if a faulty process claims to have received a specific message from a correct process, then that claim can be verified using authentication, based on unforgeable signatures.

Byzantine or malicious failure

- In this model, a process may exhibit any arbitrary behavior and no authentication techniques are applicable to verify any claims made.

11. FAILURE MODELS

Communication failure models

Crash failure

A properly functioning link may stop carrying messages from some instant thenceforth.

Omission failures

A link carries some messages but not the others sent on it.

Byzantine failures

A link can exhibit any arbitrary behavior, including creating spurious messages and modifying the messages sent on it.

12. WAIT-FREE ALGORITHMS

- A **wait-free algorithm** is an algorithm that can execute (synchronization operations) in an $n-1$ -process fault tolerant manner, i.e., it is resilient to $n-1$ process failures
- Thus, if an algorithm is wait-free, then the (synchronization) operations of any process must complete in a bounded number of steps irrespective of the failures of all the other processes.
- Wait-free algorithms offer a very high degree of robustness.
- Designing a wait-free algorithm is usually very expensive and may not even be possible for some synchronization problems,

13. COMMUNICATION CHANNELS

- **Communication channels** are normally first-in first-out queues (FIFO).
- At the network layer, this property may not be satisfied, giving non-FIFO channels.

Elementary Graph Algorithms

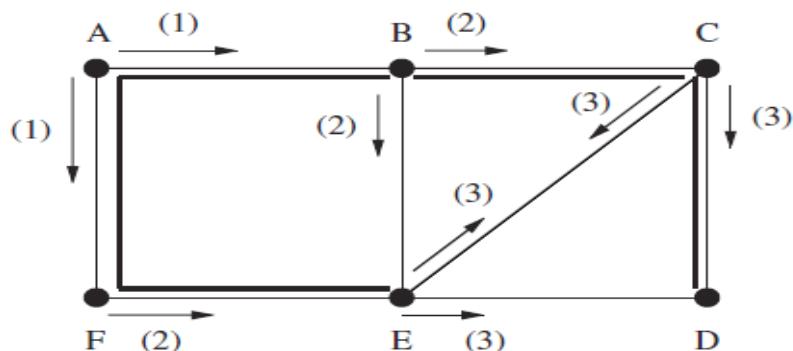
1. Synchronous Single-initiator Spanning Tree Algorithm Using Flooding
2. Asynchronous Single-initiator Spanning Tree Algorithm Using Flooding
3. Asynchronous Concurrent-initiator Spanning Tree Algorithm Using Flooding
4. Asynchronous Concurrent-initiator Depth First Search Spanning Tree Algorithm
5. Broadcast And Convergecast On A Tree
6. Single Source Shortest Path Algorithm: Synchronous Bellman–ford
7. Distance Vector Routing
8. Single Source Shortest Path Algorithm: Asynchronous Bellman–ford
9. All Sources Shortest Paths: Asynchronous Distributed Floyd–warshall
10. Asynchronous And Synchronous Constrained Flooding (W/O A Spanning Tree)
11. Minimum-weight Spanning Tree (MST) Algorithm In A Synchronous System
12. Minimum-weight Spanning Tree (MST) In An Asynchronous System

1. SYNCHRONOUS SINGLE-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

- The code for all processes is not only symmetrical, but also proceeds in rounds.
- This algorithm assumes a designated root node, **root**, which initiates the algorithm.
- The root initiates a flooding of QUERY messages in the graph to identify tree edges.
- The parent of a node is that node from which a QUERY is first received; if multiple QUERYs are received in the same round, one of the senders is randomly chosen as the parent.

1. SYNCHRONOUS SINGLE-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

- Node A as initiator at the end of round 2, E receives a QUERY from B and F and randomly chooses F as the parent.
- A total of nine QUERY messages are sent in the network which has eight links.



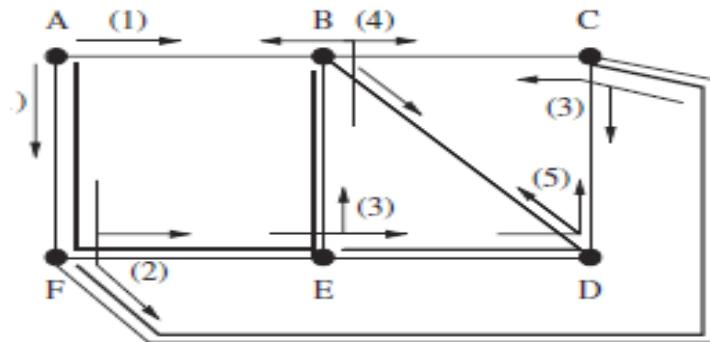
2. ASYNCHRONOUS SINGLE-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

- This algorithm assumes a designated root node which initiates the algorithm.
- The root initiates a flooding of QUERY messages in the graph to identify tree edges.
- The parent of a node is that node from which a QUERY is first received; an ACCEPT message is sent in response to such a QUERY.
- Other QUERY messages received are replied to by a REJECT message.
- Each node terminates its algorithm when it has received from all its non-parent neighbors a response to the QUERY sent to them.

2. ASYNCHRONOUS SINGLE-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

- In this asynchronous system, there is no bound on the time it takes to propagate
- Each node here needs to track its neighbors to determine which nodes are its children and which nodes are not.
- This tracking is necessary in order to know when to terminate.
- After sending QUERY messages on the outgoing links, the sender needs to know how long to keep waiting.
- This is accomplished by requiring each node to return an “acknowledgement” for each QUERY it receives. uses two Messages types – called as ACCEPT (+ ack) and REJECT (- ack) – besides the QUERY to distinguish between the child nodes and non-child nodes.

2. ASYNCHRONOUS SINGLE-INITIATOR SPANNING TREE ALGORITHM USING FLOODING



2. ASYNCHRONOUS SINGLE-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

1. A sends a QUERY to B and F.
2. F receives QUERY from A and determines that AF is a *tree edge*. F forwards the QUERY to E and C.
3. E receives a QUERY from F and determines that FE is a *tree edge*. E forwards the QUERY to B and D. C receives a QUERY from F and determines that FC is a *tree edge*. C forwards the QUERY to B and D.
4. B receives a QUERY from E and determines that EB is a *tree edge*. B forwards the QUERY to A, C, and D.
5. D receives a QUERY from E and determines that ED is a *tree edge*. D forwards the QUERY to B and C.

3. ASYNCHRONOUS CONCURRENT-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

- We modify Algorithm 2 by assuming that any node may spontaneously initiate the spanning tree algorithm provided it has not already been invoked locally due to the receipt of a QUERY message.
- The crucial problem to handle is that of dealing with concurrent initiations, where two or more processes that are not yet participating in the algorithm initiate the algorithm concurrently.
- As the objective is to construct a single spanning tree, two options seem available when concurrent initiations are detected.
- Note that even though there can be multiple concurrent initiations, along any single edge, only two concurrent initiations will be detected.

3. ASYNCHRONOUS CONCURRENT-INITIATOR SPANNING TREE ALGORITHM USING FLOODING

Design 1

- When two concurrent initiations are detected by two adjacent nodes that have sent a QUERY from different initiations to each other, the two partially computed spanning trees can be merged.

Design 2

- Suppress the instance initiated by one root and continue the instance initiated by the other root, based on some rule such as tie-breaking using the processor identifier.

4. ASYNCHRONOUS CONCURRENT-INITIATOR DEPTH FIRST SEARCH SPANNING TREE ALGORITHM

- This algorithm assumes that any node may spontaneously initiate the spanning tree algorithm provided it has not already been invoked locally due to the receipt of a QUERY message.
- It differs from Algorithm - 3 in that it is based on a depth-first search (DFS) of the graph to identify the spanning tree.
- The algorithm should handle concurrent initiations.
- The parent of each node is that node from which a QUERY is first received; an ACCEPT message is sent in response to such a QUERY.
- Other QUERY messages received are replied to by a REJECT message.

5. BROADCAST AND CONVERGECAST ON A TREE

- A spanning tree is useful for distributing (via a broadcast) and collecting (via a convergecast) information to/from all the nodes.
- A generic graph with a spanning tree, and the convergecast and broadcast operations are illustrated

5. BROADCAST AND CONVERGECAST ON A TREE

A *broadcast algorithm* on a spanning tree can be specified by two rules:

BC1: The root sends the information to be broadcast to all its children. Terminate.

BC2: When a (non root) node receives information from its parent, it copies it and forwards it to its children. Terminate.

5. BROADCAST AND CONVERGECAST ON A TREE

- A *convergecast algorithm* collects information from all the nodes at the root node in order to compute some *global function*.
- It is initiated by the leaf nodes of the tree, usually in response to receiving a request sent by the root using a broadcast.

5. BROADCAST AND CONVERGECAST ON A TREE

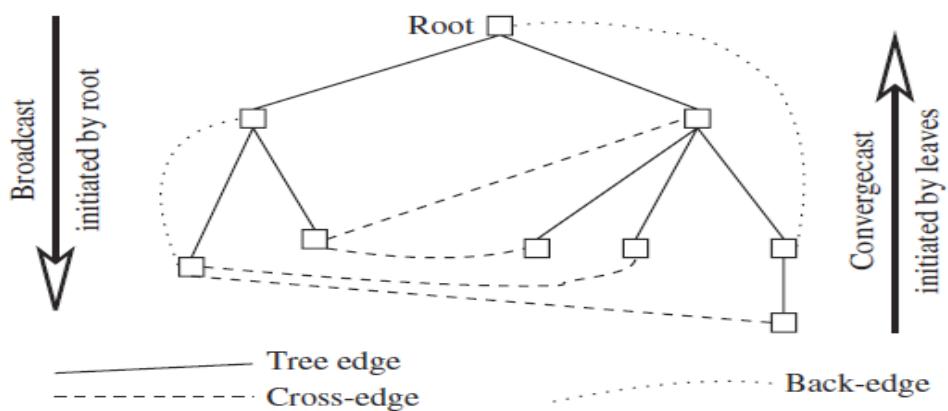
The algorithm is specified as follows:

CVC1: Leaf node sends its report to its parent. Terminate.

CVC2: At a nonleaf node that is not the root: When a report is received from all the child nodes, the collective report is sent to the parent. Terminate.

CVC3: At the root: When a report is received from all the child nodes, the global function is evaluated using the reports. Terminate.

5. BROADCAST AND CONVERGECAST ON A TREE



6. SINGLE SOURCE SHORTEST PATH ALGORITHM: SYNCHRONOUS BELLMAN–FORD

- Given a weighted graph, with potentially unidirectional links, representing the network topology, the Bellman–Ford sequential shortest path algorithm finds the shortest path from a given node, to all other nodes.
- The algorithm is correct when there are no cyclic paths having negative weight.
- A synchronous distributed algorithm to compute the shortest path is assumed that the topology is not known to any process; rather, each process can communicate only with its neighbors and is aware of only the incident links and their weights.
- It is also assumed that the processes know the number of nodes $N = n$, i.e., the algorithm is not uniform.
- This assumption on n is required for termination.

7. DISTANCE VECTOR ROUTING

When the network graph is dynamically changing, as in a real communication network wherein the link weights model the delays or loads on the links, the shortest paths are required for routing. The classic distance vector routing algorithm used in the ARPANET up to 1980, is based on the above synchronous algorithm (Algorithm 5) and requires the following changes:

7. DISTANCE VECTOR ROUTING

- The outer **for** loop runs indefinitely, and the length and parent variables never stabilize, because of the dynamic nature of the system.
- The variable length is replaced by array LENGTH [1..n], where LENGTH[k] denotes the length measured with node k as source/root.
- The LENGTH vector is also included on each UPDATE message. Now, the kth component of the LENGTH received from node m indicates the length of the shortest path from m to the root k. For each destination k, the triangle inequality of the Bellman–Ford algorithm is applied over all the LENGTH vectors received in a round.

7. DISTANCE VECTOR ROUTING

- The variable parent is replaced by array PARENT[1..n], where PARENT[k] denotes the next hop to which to route a packet destined for k.
- The array PARENT serves as the routing table.
- The processes exchange their distance vectors periodically over a network that is essentially asynchronous.

7. DISTANCE VECTOR ROUTING

- If a message does not arrive within the period, the algorithm assumes a default value, and moves to the next round.
- This makes it virtually synchronous.
- Besides, if the period between exchanges is assumed to be much larger than the propagation time from a neighbor and the processing time for the received message, the algorithm is effectively synchronous.

8. SINGLE SOURCE SHORTEST PATH ALGORITHM: ASYNCHRONOUS BELLMAN–FORD

- The asynchronous version of the Bellman–Ford algorithm is assumed that there are no negative weight cycles in N L.
- The algorithm does not give the termination condition for the nodes.
- Each node knows when the length of the shortest path to itself has been computed.
- This algorithm, unfortunately, has been shown to have an exponential number of messages and exponential time complexity
- If all links are assumed to have equal weight, the algorithm that computes the shortest path effectively computes the minimum-hop path; the minimum hop routing tables to all destinations are computed

9. ALL SOURCES SHORTEST PATHS: ASYNCHRONOUS DISTRIBUTED FLOYD WARSHALL

- The Floyd–Warshall algorithm computes all-pairs shortest paths in a graph in which there are no negative weight cycles.
- The centralized algorithm uses $n \times n$ matrices LENGTH and VIA:

LENGTH[i, j] is the length of the shortest path from i to j . $LENGTH[i, j]$ is initialized to the initial known conditions: (i) $weight_{i,j}$ if i and j are neighbors, (ii) 0 if $i = j$, and (iii) ∞ otherwise.

VIA[i, j] is the first hop on the shortest path from i to j . $VIA[i, j]$ is initialized to the initial known conditions: (i) j if i and j are neighbors, (ii) 0 if $i = j$, and (iii) ∞ otherwise.

10. ASYNCHRONOUS AND SYNCHRONOUS CONSTRAINED FLOODING (W/O A SPANNING TREE)

Asynchronous algorithm

- This algorithm allows any process to initiate a broadcast via (constrained) flooding along the edges of the graph.
- It is assumed that all channels are FIFO.
- Duplicates are detected by using sequence numbers. Each process uses the SEQNO[1..n] vector, where SEQNO[k] tracks the latest sequence number of the update initiated by process k.
- If the sequence number on a newly arrived message is not greater than the sequence numbers already seen for that initiator, the message is simply discarded; otherwise, it is flooded on all other outgoing links.
- This mechanism is used by the link state routing protocol in the Internet to distribute any updates about the link loads and the network topology.

10. ASYNCHRONOUS AND SYNCHRONOUS CONSTRAINED FLOODING (W/O A SPANNING TREE)

Synchronous algorithm

- This algorithm allows all processes to flood a local value throughout the network.
- The local array STATEVEC[1..n] is such that STATEVEC[k] is the estimate of the local value of process k.
- After d number of rounds, it is guaranteed that the local value of each process has propagated throughout the network.

11. MINIMUM-WEIGHT SPANNING TREE (MST) ALGORITHM IN A SYNCHRONOUS SYSTEM

- A minimum-weight spanning tree (MST) minimizes the cost of transmission from any node to any other node in the graph.

Kruskal's algorithm

Kruskal's algorithm begins with a forest of graph components. In each iteration, it identifies the minimum-weight edge that connects two different components, and uses this edge to merge two components. This continues until all the components are merged into a single component.

11. MINIMUM-WEIGHT SPANNING TREE (MST) ALGORITHM IN A SYNCHRONOUS SYSTEM

Prim's algorithm and Dijkstra's algorithm

In Prim's algorithm and Dijkstra's algorithm, a single-node component is selected. In each iteration, a minimum-weight edge incident on the component is identified, and the component expands to include that edge and the node at the other end of that edge. After $n-1$ iterations, all the nodes are included. The MST is defined by the edges that are identified in each iteration to expand the initial component.

12. MINIMUM-WEIGHT SPANNING TREE (MST) IN AN ASYNCHRONOUS SYSTEM

There are two approaches to designing the asynchronous algorithm.

- In the first approach, the synchronous GHS algorithm is *simulated* in an asynchronous setting. In such a simulation, the same synchronous algorithm is run, but is augmented by additional protocol steps and control messages to provide the synchronicity.

12. MINIMUM-WEIGHT SPANNING TREE (MST) IN AN ASYNCHRONOUS SYSTEM

- The second approach to designing the asynchronous is to directly address all the difficulties that arise due to lack of synchrony.
- The original asynchronous algorithm uses this approach even though it is patterned along the synchronous algorithm.
- By carefully engineering the asynchronous algorithm, it achieves the same message complexity as the synchronous algorithm and a time complexity.

SYNCHRONIZERS

General observations on synchronous and asynchronous algorithms

- From the spanning tree algorithms, shortest path routing algorithms, constrained flooding algorithms, and the MST algorithms, it can be observed that it is much more difficult to design the algorithm for an asynchronous system, than for a synchronous system.
- This can be generalized to all algorithms, with few exceptions.

SYNCHRONIZERS

- Given that typical distributed systems are asynchronous, the logical question to address is whether there is a general technique to convert an algorithm designed for a synchronous system, to run on an asynchronous system.
- The generic class of transformation algorithms to run synchronous algorithms on asynchronous systems are called **SYNCHRONIZERS**

MAXIMAL INDEPENDENT SET (MIS)

- The maximal independent set problem requires that adjacent nodes must not be chosen.
- This has application in wireless broadcast where it is required that transmitters must not broadcast on the same frequency within range of each other.
- More generally, for any shared resources to allow a maximum concurrent use while avoiding interference or conflicting use, a maximal independent set is required.
- Computing a maximal independent set in a distributed manner is challenging.
- The problem becomes further interesting when a maximal independent set must be maintained when processes join and leave, and links can go down, or new links between existing nodes can be established.

CONNECTED DOMINATING SET

- The connected dominating set can form a backbone along which a broadcast can be performed.
- All nodes are guaranteed to be within range of the backbone and can hence receive the broadcast.
- The set is thus useful for routing, particularly in the wide-area network and also in wireless networks.
- A simple heuristic is to create a spanning tree and delete the edges to the leaf nodes to get a CDS.
- Another heuristic is to create an MIS and add edges to create a CDS.
- However, designing an algorithm with a low approximation factor is non-trivial.

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

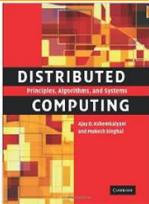
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



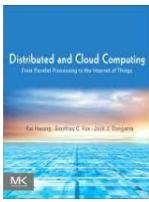
CS-5 : Message ordering and Termination detection

[T1: Chap – 6]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

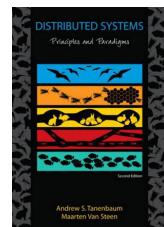


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 5

Message ordering and Termination detection

- In a distributed system, message ordering plays an important role on making the system consistent or stable.
- The order of messages sent should match (happen prior) the order of its' receipt in a group communication application implemented in a distributed system.
- This module will discuss ways to achieve various message ordering schemes.
- Also, the approaches for detecting termination of a distributed computation will be discussed.

Contact Session – 5

Message ordering and Termination detection

- Message ordering paradigms
- Group Communication
- Protocols for ensuring Causal order of messages

Presentation Overview

- Introduction
- Message ordering paradigms
 - Asynchronous executions
 - FIFO executions
 - Causally ordered (CO) executions
 - Synchronous execution (SYNC)
- Group communication
- Causal order (CO)

Introduction

- Inter-process communication via message-passing is at the core of any distributed system.
- We will study non-FIFO, FIFO, causal order, and synchronous order communication paradigms for ordering messages

Message ordering paradigms

- The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program.
- Distributed program logic greatly depends on this order of delivery.
- To simplify the task of the programmer, programming languages in conjunction with the middleware provide certain well-defined message delivery behavior.
- The programmer can then code the program logic with respect to this behavior.

Message ordering paradigms

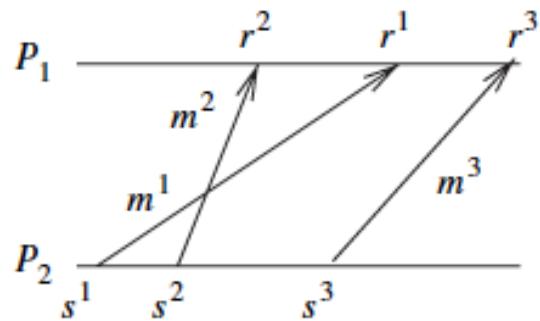
- Several orderings on messages have been defined:
 - Non-FIFO
 - FIFO
 - Causal order
 - Synchronous order
- There is a natural hierarchy among these orderings.
- This hierarchy represents a trade-off between concurrency and ease of use and implementation

Asynchronous executions

- An asynchronous execution (or A-execution) is an execution for which the causality relation is a partial order
- On any logical link between two nodes in the system, messages may be delivered in any order, not necessarily first-in first-out.
- Such executions are also known as non-FIFO executions .
- Although each physical link typically delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logical link may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link.
- As an example, the mode of ordering at the Network Layer in connectionless networks such as IPv4 is non-FIFO

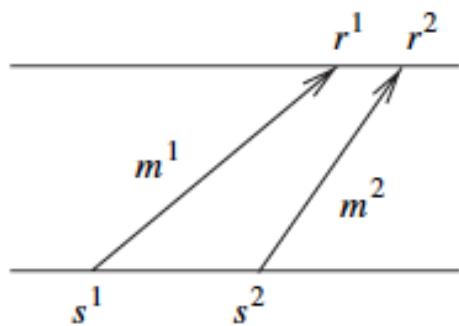
Asynchronous executions

illustrates an A -execution under non-FIFO ordering



Asynchronous executions

Illustrates an A -execution under FIFO ordering



FIFO executions

- On any logical link in the system, messages are necessarily delivered in the order in which they are sent.
- Although the logical link is inherently non-FIFO, most network protocols provide a connection-oriented service at the transport layer.
- Therefore, FIFO logical channels can be realistically assumed when designing distributed algorithms.

FIFO executions

- A simple algorithm to implement a FIFO logical channel over a non-FIFO channel would use a separate numbering scheme to sequence the messages on each logical channel.
- The sender assigns and appends a `<sequence_num, connection_id>` tuple to each message.
- The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence.

Causally ordered (CO) executions

- If two send events s and s' are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events r and r' occur in the same order at all common destinations.
- Note that if s and s' are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false.

Causally ordered (CO) executions

- Causal ordering is a vital tool for thinking about distributed systems.
- The fundamental property of 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.

Causally ordered (CO) executions

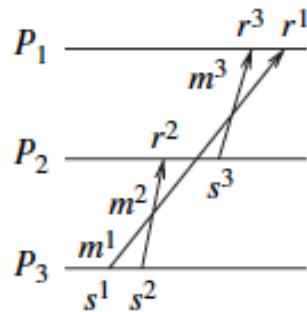
- For example, because of this property it is impossible for two computers communicating over a network to agree on the exact time.
- You can send me a message saying “it is now 10:00:00” but I don’t know how long it took for that message to arrive.
- We can send messages back and forth all day but we 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.

Causally ordered (CO) executions

- 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.

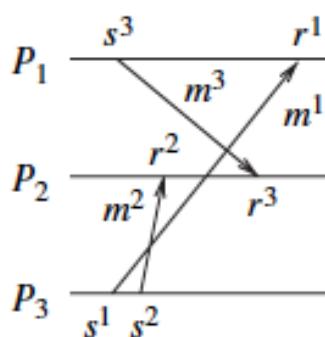
Causally ordered (CO) executions

An execution that violates CO because $s_1 \prec s_3$ and at the common destination P_1 , we have $r_3 \prec r_1$



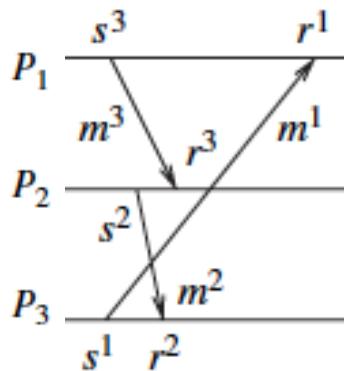
Causally ordered (CO) executions

An execution that satisfies CO. Only s_1 and s_2 are related by causality but the destinations of the corresponding messages are different.



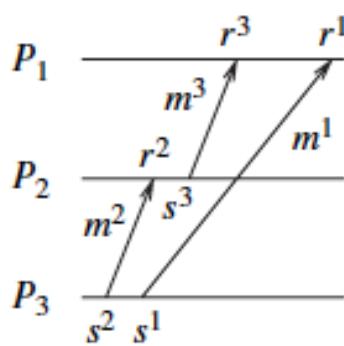
Causally ordered (CO) executions

An execution that satisfies CO. No send events are related by causality



Causally ordered (CO) executions

An execution that satisfies CO. s2 and s1 are related by causality but the destinations of the corresponding messages are different. Similarly for s2 and s3



Causally ordered (CO) executions

- Causal order is useful for applications requiring updates to shared data, implementing distributed shared memory, and fair resource allocation such as granting of requests for distributed mutual exclusion.

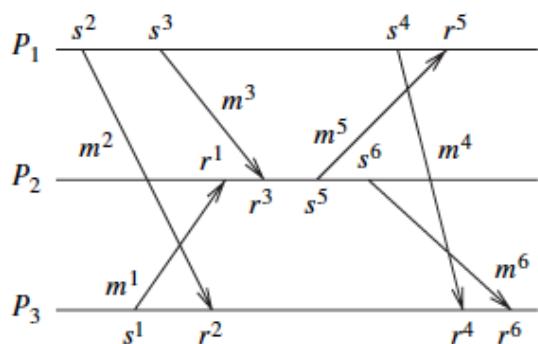
Causally ordered (CO) executions

- To implement CO, we distinguish between the arrival of a message and its delivery.
- A message m that arrives in the local OS buffer at P_i may have to be delayed until the messages that were sent to P_i causally before m was sent (the “overtaken” messages) have arrived and are processed by the application.
- The delayed message m is then given to the application for processing.
- The event of an application processing an arrived message is referred to as a delivery event (instead of as a receive event) for emphasis.

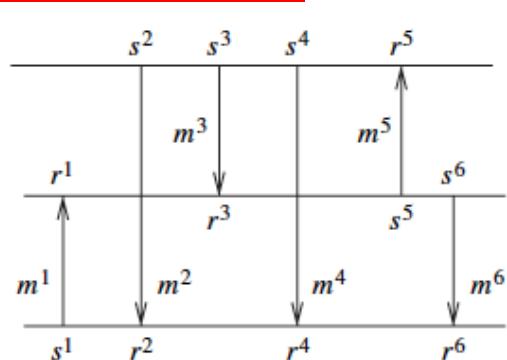
Synchronous execution (SYNC)

- When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order.
- As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occurring instantaneously and atomically.

Synchronous execution (SYNC)



(a)



(b)

Illustration of a synchronous communication.

- Execution in an asynchronous system.
- Equivalent instantaneous communication.

Group communication

- Processes across a distributed system cooperate to solve a joint task.
- Often, they need to communicate with each other as a group, and therefore there needs to be support for group communication .
- A message broadcast is the sending of a message to all members in the distributed system.
- The notion of a system can be confined only to those sites/processes participating in the joint application.

Group communication

- Refining the notion of broadcasting , there is multicasting wherein a message is sent to a certain subset, identified as a group , of the processes in the system.
- At the other extreme is unicasting , which is the familiar point-to-point message communication.

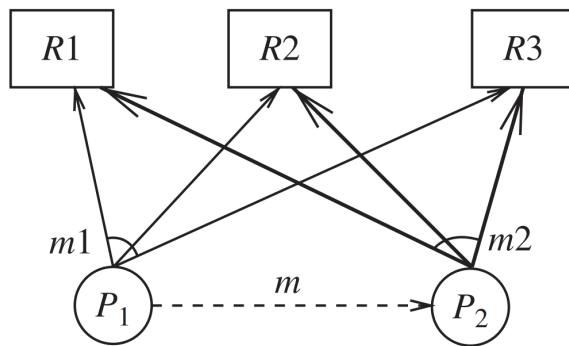
Group communication

- If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a closed group algorithm.
- If the sender of the multicast can be outside the destination group, the multicast algorithm is said to be an open group algorithm

Causal order (CO)

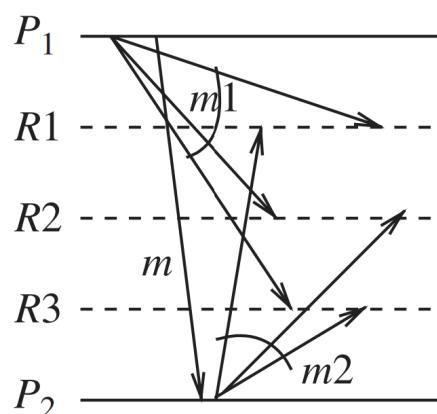
- Causal order has many applications such as updating replicated data, allocating requests in a fair manner, and synchronizing multimedia streams.
- The use of causal order in updating replicas of a data item in the system.
- Consider two processes P1 and P2 that issue updates to the three replicas R1d, R2d, and R3d of data item d.
- Message m creates a causality between send(m1) and send(m2).
- If P2 issues its update causally after P1 issued its update, then P2's update should be seen by the replicas after they see P1's update, in order to preserve the semantics of the application.
- In this case, CO is satisfied

Causal order (CO)



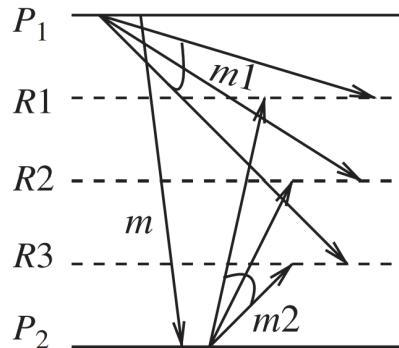
Causal order (CO)

- $R1$ sees $P2$'s update first, while $R2$ and $R3$ see $P1$'s update first. Here, CO is violated.



Causal order (CO)

- All replicas see P_2 's update first.
- CO is still violated If message m did not exist, then the executions shown would satisfy CO.



THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

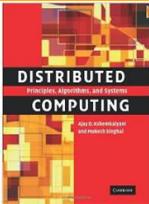


BITS Pilani
Pilani | Dubai | Goa | Hyderabad

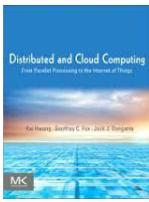
CS-6 : Message ordering and Termination detection

[T1: Chap – 6 & 7]

Text and References



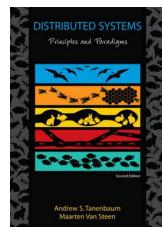
T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).



R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.



R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall

Objective of Module – 5

Message ordering and Termination detection

- In a distributed system, message ordering plays an important role on making the system consistent or stable.
- The order of messages sent should match (happen prior) the order of its' receipt in a group communication application implemented in a distributed system.
- This module will discuss ways to achieve various message ordering schemes.
- Also, the approaches for detecting termination of a distributed computation will be discussed.

Contact Session – 6

Message ordering and Termination detection

- Total order
- Application-level multicast
- Distributed Multicast and Steiner Trees - optional
- Termination detection using distributed snapshots
- Termination detection using weight throwing
- A spanning-tree based termination detection algorithm

Presentation Overview

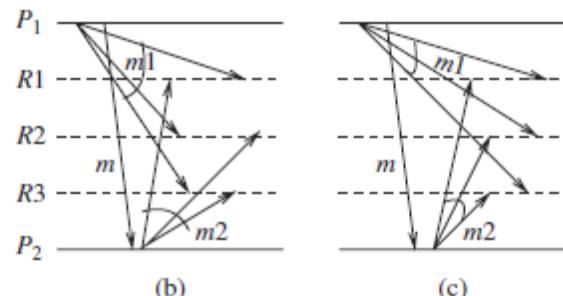
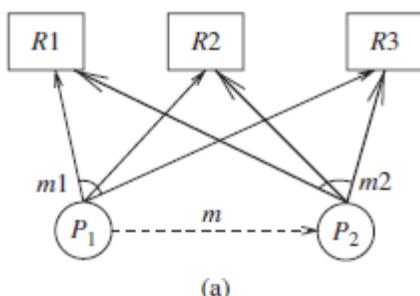
- Total order
- A nomenclature for multicast
- Application-level multicast
 - Classification of application-level multicast algorithms
 - Communication history-based algorithms
 - Privilege-based algorithms
 - Moving sequencer algorithms
 - Fixed sequencer algorithms
 - Destination agreement algorithms
- Termination detection - Introduction
- Termination detection using distributed snapshots
- Termination detection by weight throwing
- A spanning-tree-based termination detection algorithm

Total order

- Consider the example of updates to replicated data
- As the replicas are of just one data item d, it would be logical to expect that all replicas see the updates in the same order, whether or not the issuing of the updates are causally related.
- This way, the issue of coherence and consistency of the replica values goes away.
- Such a replicated system would still be useful for fault-tolerance, as well as for easy availability for “read” operations.
- Total order, which requires that all messages be received in the same order by the recipients of the messages

Total order

- The execution in Figure (b) does not satisfy total order. Even if the message m did not exist, total order would not be satisfied.
- The execution in Figure (c) satisfies total order



Three-phase distributed algorithm

- A distributed algorithm that enforces total and causal order for closed groups
- The three phases of the algorithm are first described from the
 - viewpoint of the sender
 - viewpoint of the receiver

Sender

Phase 1

- In the first phase, a process multicasts the message M with a locally unique tag and the local timestamp to the group members.

Phase 2

- In the second phase, the sender process awaits a reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M.
- Once all expected replies are received, the process computes the maximum of the proposed timestamps for M, and uses the maximum as the final timestamp.

Phase 3

In the third phase, the process multicasts the final timestamp to the group

Receivers

Phase 1

- In the first phase, the receiver receives the message with a tentative/proposed timestamp.
- It updates the variable *priority* that tracks the highest proposed timestamp then revises the proposed timestamp to the *priority*, and places the message with its tag and the revised timestamp at the tail of the queue
- In the queue, the entry is marked as undeliverable

Receivers

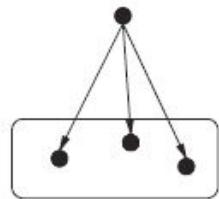
Phase 2

- In the second phase, the receiver sends the revised timestamp (and the tag) back to the sender
- The receiver then waits in a non-blocking manner for the final timestamp

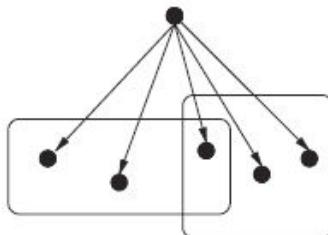
Phase 3

- In the third phase, the final timestamp is received from the multicaster
- The corresponding message entry in temp_Q is identified using the tag and is marked as deliverable after the revised timestamp is overwritten by the final timestamp
- The queue is then resorted using the timestamp field of the entries as the key

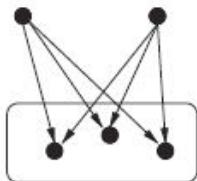
Nomenclature for multicast



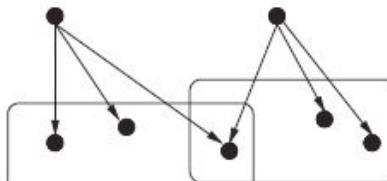
(a) Single source single group (SSSG)



(c) Single source multiple groups (SSMG)



(b) Multiple sources single group (MSSG)



(d) Multiple sources multiple groups (MSMG)

Classification of application-level multicast algorithms

- The most general scenario allows each process to multicast to an arbitrary and dynamically changing group of processes at each step.
- As this generality incurs more overhead, algorithms implemented on real systems tend to be more “centralized” in one sense or another
- Many multicast protocols have been developed and deployed, but they can all be classified as belonging to one of the following five classes.

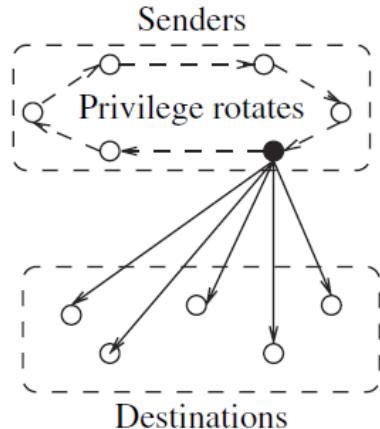
Communication history-based algorithms

- Algorithms in this class use a part of the communication history to guarantee ordering requirements
- They do not need to track separate groups, and hence work for open-group multicasts.
- Lamport's algorithm, wherein messages are assigned scalar timestamps and a process can deliver a message only when it knows that no other message with a lower timestamp can be multicast, belongs to this class.

Privilege-based algorithms

- A token circulates among the sender processes.
- The token carries the sequence number for the next message to be multicast, and only the token-holder can multicast.
- After a multicast send event, the sequence number is updated.
- Destination processes deliver messages in the order of increasing sequence numbers.
- Senders need to know the other senders, hence closed groups are assumed.
- Such algorithms can provide total ordering, as well as causal ordering using a closed group configuration
- Such algorithms are not scalable because they do not permit concurrent send events. Hence they are of limited use in large systems.

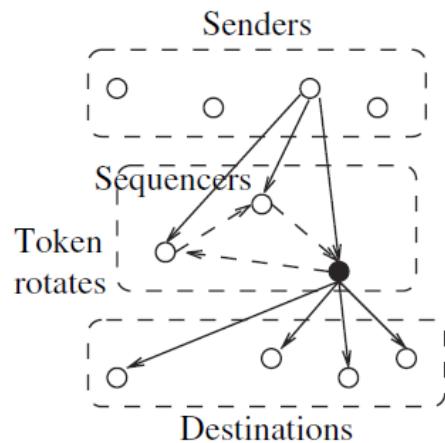
Privilege-based algorithms



Moving sequencer algorithms

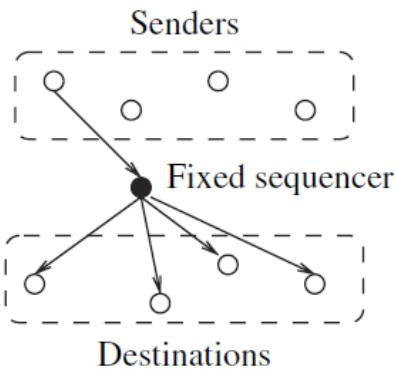
- To multicast a message, the sender sends the message to all the sequencers.
- Sequencers circulate a token among themselves.
- The token carries a sequence number and a list of all the messages for which a sequence number has already been assigned – such messages have been sent already.
- When a sequencer receives the token, it assigns a sequence number to all received but unsequenced messages.
- It then sends the newly sequenced messages to the destinations, inserts these messages in to the token list, and passes the token to the next sequencer.
- Destination processes deliver the messages received in the order of increasing sequence number. Moving sequencer algorithms guarantee total ordering.

Moving sequencer algorithms



Fixed sequencer algorithms

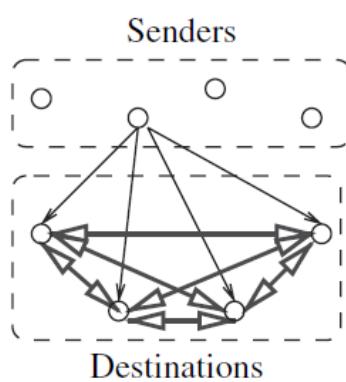
- There is a single sequencer which makes this class of algorithms essentially centralized



Destination agreement algorithms

- In this class of algorithms, the destinations receive the messages with some limited ordering information.
- They then exchange information among themselves to define an order.
- There are two sub-classes here:
 - the first sub-class uses timestamps
 - the second sub-class uses an agreement or “consensus” protocol among the processes

Destination agreement algorithms



Termination detection - Introduction

- In distributed processing systems, a problem is typically solved in a distributed manner with the cooperation of a number of processes.
- In such an environment, inferring if a distributed computation has ended is essential so that the results produced by the computation can be used.
- Also, in some applications, the problem to be solved is divided into many subproblems, and the execution of a subproblem cannot begin until the execution of the previous subproblem is complete.
- Hence, it is necessary to determine when the execution of a particular subproblem has ended so that the execution of the next subproblem may begin.
- Therefore, a fundamental problem in distributed systems is to determine if a distributed computation has terminated.

Termination detection - Introduction

- The detection of the termination of a distributed computation is non-trivial since no process has complete knowledge of the global state, and global time does not exist.
- A distributed computation is considered to be globally terminated if every process is locally terminated and there is no message in transit between any processes.
- A “locally terminated” state is a state in which a process has finished its computation and will not restart any action unless it receives a message

Termination detection - Introduction

- When we are interested in inferring when the underlying computation has ended, a termination detection algorithm is used for this purpose.
- In such situations, there are two distributed computations taking place in the distributed system, namely, the *underlying computation* and the *termination detection algorithm*.
- Messages used in the underlying computation are called *basic* messages, and messages used for the purpose of termination detection are called *control* messages.

Termination detection - Introduction

A termination detection (TD) algorithm must ensure the following:

- Execution of a TD algorithm cannot indefinitely delay the underlying computation; that is, execution of the termination detection algorithm must not freeze the underlying computation.
- The termination detection algorithm must not require addition of new communication channels between processes

Termination detection using distributed snapshots

- The algorithm uses the fact that a consistent snapshot of a distributed system captures stable properties.
- Termination of a distributed computation is a stable property.
- Thus, if a consistent snapshot of a distributed computation is taken after the distributed computation has terminated, the snapshot will capture the termination of the computation.
- The algorithm assumes that there is a logical bidirectional communication channel between every pair of processes.
- Communication channels are reliable but non-FIFO. Message delay is arbitrary but finite.

Termination detection using distributed snapshots

- The main idea behind the algorithm is when a computation terminates, there must exist a unique process which became idle last.
- When a process goes from active to idle, it issues a request to all other processes to take a local snapshot, and also requests itself to take a local snapshot.
- When a process receives the request, if it agrees that the requester became idle before itself, it grants the request by taking a local snapshot for the request.
- A request is said to be successful if all processes have taken a local snapshot for it.

Termination detection using distributed snapshots

- The requester or any external agent may collect all the local snapshots of a request.
- If a request is successful, a global snapshot of the request can thus be obtained and the recorded state will indicate termination of the computation, viz., in the recorded snapshot, all the processes are idle and there is no message in transit to any of the processes.

Termination detection by weight throwing

- In termination detection by weight throwing, a process called controlling agent1 monitors the computation.
- A communication channel exists between each of the processes and the controlling agent and also between every pair of processes

Termination detection by weight throwing

- Initially, all processes are in the idle state.
- The weight at each process is zero and the weight at the controlling agent is 1.
- The computation starts when the controlling agent sends a basic message to one of the processes.
- The process becomes active and the computation starts.
- A non-zero weight W ($0 < W \leq 1$) is assigned to each process in the active state and to each message in transit in the following manner:

Termination detection by weight throwing

- When a process sends a message, it sends a part of its weight in the message.
- When a process receives a message, it adds the weight received in the message to its weight.
- Thus, the sum of weights on all the processes and on all the messages in transit is always 1.
- When a process becomes passive, it sends its weight to the controlling agent in a control message, which the controlling agent adds to its weight.
- The controlling agent concludes termination if its weight becomes 1.

A spanning-tree-based termination detection algorithm

- The algorithm assumes there are N processes P_i , $0 \leq i \leq N$, which are modeled as the nodes i , $0 \leq i \leq N$, of a fixed connected undirected graph.
- The edges of the graph represent the communication channels, through which a process sends messages to neighboring processes in the graph.
- The algorithm uses a fixed spanning tree of the graph with process P_0 at its root which is responsible for termination detection.
- Process P_0 communicates with other processes to determine their states and the messages used for this purpose are called signals.

A spanning-tree-based termination detection algorithm

- All leaf nodes report to their parents, if they have terminated.
- A parent node will similarly report to its parent when it has completed processing and all of its immediate children have terminated, and so on.
- The root concludes that termination has occurred, if it has terminated and all of its immediate children have also terminated.

A spanning-tree-based termination detection algorithm

- The termination detection algorithm generates two waves of signals moving inward and outward through the spanning tree.
- Initially, a contracting wave of signals, called tokens , moves inward from leaves to the root.
- If this token wave reaches the root without discovering that termination has occurred, the root initiates a second outward wave of repeat signals.
- As this repeat wave reaches leaves, the token wave gradually forms and starts moving inward again.
- This sequence of events is repeated until the termination is detected

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

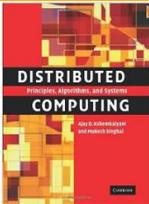
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



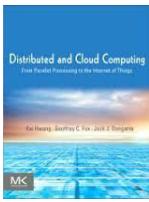
CS-7 : Distributed Mutual Exclusion & Deadlock detection

[T1: Chap – 9]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

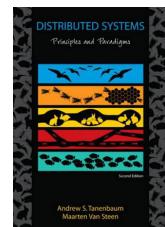


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 6

Distributed Mutual Exclusion & Deadlock detection

- In a distributed system when multiple entities compete for a shared resource, the access to this shared resource has to be serialized (or coordinated).
- This module will cover different assertion based, and tree based distributed algorithms to implement DME.
- Deadlocks are very common in distributed computing systems, and this module will discuss ways to detect this deadlock and resolve them

Contact Session – 7

Distributed Mutual Exclusion & Deadlock detection

- Introduction and Preliminaries
- Assertion based: Lamport's algorithm, and Ricart-Agrawala's algorithm
- Assertion based: Maekawa's algorithm

Presentation Overview

- Mutual Exclusion
- Critical Section(CS)
- Preliminaries
 - System model
 - Requirements of mutual exclusion algorithms
 - Performance metrics
- Mutual Exclusion
 - Token-based approach
 - Non-token-based approach
 - Quorum-based approach
- Lamport's algorithm
- Ricart–Agrawala algorithm
- Maekawa's algorithm

Mutual Exclusion

- In computer science, **mutual exclusion** refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time
- It is a basic requirement in concurrency control, to prevent race conditions.
- Here, a critical section refers to a period when the process accesses a shared resource, such as shared memory

Critical Section(CS)

- In concurrent programming, a **critical section** is a part of a multi-process program that may not be concurrently executed by more than one of the program's processes/threads
- In other words, it is a piece of program that requires mutual exclusion of access.
- Typically, the critical section accesses a shared resource (data structure or device)
- A critical section may consist of multiple discontiguous parts of the program's code.

Critical Section(CS)

- For example, one part of a program might read from a file that another part wishes to modify.
- These parts together form a single critical section, since simultaneous readings and modifications may interfere with each other.
- A critical section will usually terminate in finite time, and a thread, task, or process will have to wait for a fixed time to enter it
- Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use

Preliminaries

- Here we understand the underlying system model, and the requirements that mutual exclusion algorithms should satisfy, along with metrics used to measure the performance of mutual exclusion algorithms.

System model

- The system consists of N sites, S₁, S₂, ..., S_N.
- The process at site S_i is denoted by p_i. All these processes communicate asynchronously over an underlying communication network.
- A process wishing to enter the CS requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the CS.
- While waiting the process is not allowed to make further requests to enter the CS.

System model

- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the “requesting the CS” state, the site is blocked and cannot make further requests for the CS.
- In the “idle” state, the site is executing outside the CS.
- In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS.
- Such state is referred to as the *idle token* state. At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

Requirements of mutual exclusion algorithms

1. Safety property

- The safety property states that at any instant, only one process can execute the critical section.
- This is an essential property of a mutual exclusion algorithm.

Requirements of mutual exclusion algorithms

2. Liveness property

- This property states the absence of deadlock and starvation.
- Two or more sites should not endlessly wait for messages that will never arrive.
- In addition, a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS.
- That is, every requesting site should get an opportunity to execute the CS in finite time.

Requirements of mutual exclusion algorithms

3. Fairness

- Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS.
- In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system the time is determined by a logical clock

Performance metrics

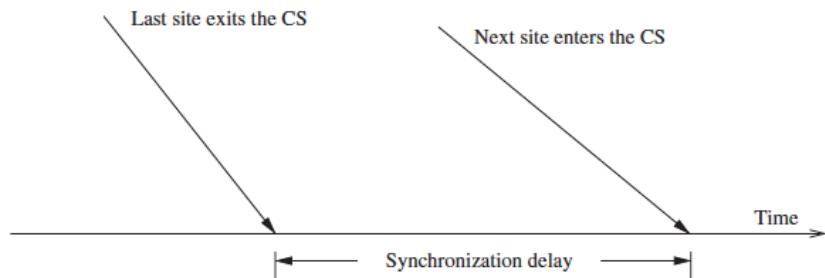
Message complexity

- This is the number of messages that are required per CS execution by a site.

Synchronization delay

- After a site leaves the CS, it is the time required and before the next site enters the CS
- Note that normally one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS.

Performance metrics

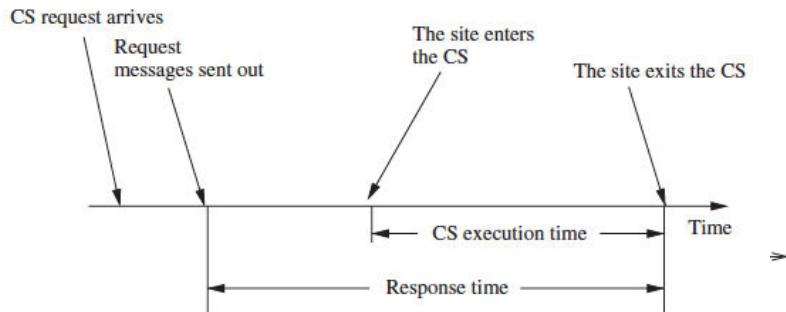


Performance metrics

Response time

- This is the time interval a request waits for its CS execution to be over after its request messages have been sent out
- Thus, response time does not include the time a request waits at a site before its request messages have been sent out.

Performance metrics



Performance metrics

System throughput

- This is the rate at which the system executes requests for the CS.
- If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation

$$\text{System throughput} = \frac{1}{(SD+E)}.$$

Mutual Exclusion

- The design of distributed mutual exclusion algorithms is complex because these algorithms have to deal with unpredictable message delays and incomplete knowledge of the system state.
- There are three basic approaches for implementing distributed mutual exclusion:
 1. Token-based approach.
 2. Non-token-based approach.
 3. Quorum-based approach.

Token-based approach

- In the token-based approach, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over.
- Mutual exclusion is ensured because the token is unique.
- The algorithms based on this approach essentially differ in the way a site carries out the search for the token.

Non-token-based approach

- In the non-token-based approach, two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
- A site enters the critical section (CS) when an assertion, defined on its local variables, becomes true.
- Mutual exclusion is enforced because the assertion becomes true only at one site at any given time.

Quorum-based approach

- In the quorum-based approach, each site requests permission to execute the CS from a subset of sites (called a quorum).
- The quorums are formed in such a way that when two sites concurrently request access to the CS, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time.

Lamport's algorithm

- Lamport developed a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme
- The algorithm is fair in the sense that a request for CS are executed in the order of their timestamps and time is determined by logical clocks.
- When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp.
- The algorithm executes CS requests in the increasing order of timestamps.
- Every site S_i keeps a queue, request_queue_i , which contains mutual exclusion requests ordered by their timestamps.

Ricart–Agrawala algorithm

- The Ricart–Agrawala algorithm assumes that the communication channels are FIFO.
- The algorithm uses two types of messages: REQUEST and REPLY.
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section.
- A process sends a REPLY message to a process to give its permission to that process.
- Processes use Lamport-style logical clocks to assign a timestamp to critical section requests. Timestamps are used to decide the priority of requests in case of conflict

Ricart–Agrawala algorithm

- If a process p_i that is waiting to execute the critical section receives a REQUEST message from process p_j , then if the priority of p_j 's request is lower, p_i defers the REPLY to p_j and sends a REPLY message to p_j only after executing the CS for its pending request.
- Otherwise, p_i sends a REPLY message to p_j immediately, provided it is currently not executing the CS.
- Thus, if several processes are requesting execution of the CS, the highest priority request succeeds in collecting all the needed REPLY messages and gets to execute the CS.

Maekawa's algorithm

- There is at least one common site between the request sets of any two sites, every pair of sites has a common site which mediates conflicts between the pair.
- A site can have only one outstanding REPLY message at any time; that is, it grants permission to an incoming request if it
- has not granted permission to some other site.
- Therefore, mutual exclusion is guaranteed.
- This algorithm requires delivery of messages to be in the order they are sent between every pair of sites.

Maekawa's algorithm`

- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm.
- Condition M3 states that the size of the requests sets of all sites must be equal, which implies that all sites should have to do an equal amount of work to invoke mutual exclusion.
- Condition M4 enforces that exactly the same number of sites should request permission from any site, which implies that all sites have “equal responsibility” in granting permission to other sites.

Maekawa's algorithm

Requesting the critical section:

- (a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i .
- (b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.

Maekawa's algorithm

Executing the critical section:

- (c) Site S_i executes the CS only after it has received a REPLY message from every site in R_i .

Maekawa's algorithm

Releasing the critical section:

- (d) After the execution of the CS is over, site S_i sends a RELEASE(i) message to every site in R_i .
- (e) When a site S_j receives a RELEASE(i) message from site S_i , it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

THANK YOU

IMP Note to Self



STOP RECORDING





BITS Pilani

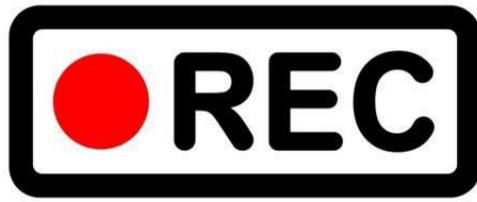
Pilani | Dubai | Goa | Hyderabad

CC ZG526

Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



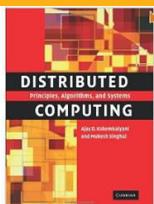
START RECORDING

CC ZG526 Distributed Computing || Week - 8 Dt: 9th Mar 2025 2 BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

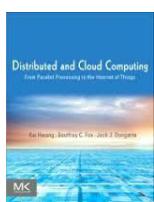
CS-7 : Distributed Mutual Exclusion & Deadlock detection

[T1: Chap – 9 & 10]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

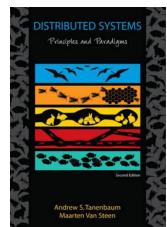


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 6

Distributed Mutual Exclusion & Deadlock detection

- In a distributed system when multiple entities compete for a shared resource, the access to this shared resource has to be serialized (or coordinated).
- This module will cover different assertion based, and tree based distributed algorithms to implement DME.
- Deadlocks are very common in distributed computing systems, and this module will discuss ways to detect this deadlock and resolve them

Contact Session – 8

Distributed Mutual Exclusion & Deadlock detection

- Token based: Suzuki-Kasami's broadcast based algorithm
- Token based: Raymond's tree based algorithm
- Models of distributed deadlock
- Chandy-Misra-Haas deadlock detection for AND model
- Chandy-Misra-Haas deadlock detection for OR model
- Deadlock resolution

Presentation Overview

- Suzuki–Kasami's broadcast algorithm
- Raymond's tree-based algorithm
- Models of distributed deadlock
- Chandy-Misra-Haas deadlock detection for AND model
- Chandy-Misra-Haas deadlock detection for OR model

Suzuki–Kasami's broadcast algorithm

- In Suzuki–Kasami's algorithm, if a site that wants to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all other sites.
- A site that possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
- If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

Suzuki–Kasami's broadcast algorithm

Although the basic idea underlying this algorithm may sound rather simple, there are two design issues that must be efficiently addressed:

1. How to distinguishing an outdated REQUEST message from a current REQUEST message
2. How to determine which site has an outstanding request for the CS

Suzuki–Kasami's broadcast algorithm

1. How to distinguishing an outdated REQUEST message from a current REQUEST message

- Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied.
- If a site cannot determine if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it.
- This will not violate the correctness, however, but it may seriously degrade the performance by wasting messages and increasing the delay at sites that are genuinely requesting the token.
- Therefore, appropriate mechanisms should be implemented to determine if a token request message is outdated.

Suzuki–Kasami's broadcast algorithm

2. How to determine which site has an outstanding request for the CS

- After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them.
- The problem is complicated because when a site S_i receives a token request message from a site S_j , site S_j may have an outstanding request for the CS.
- However, after the corresponding request for the CS has been satisfied at S_j , an issue is how to inform site S_i (and all other sites) efficiently about it.

Suzuki–Kasami's broadcast algorithm

- After executing the CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS).
- Note that Suzuki–Kasami's algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of symmetric algorithm:
“no site possesses the right to access its CS when it has not been requested.”

Raymond's tree-based algorithm

- Raymond's tree-based mutual exclusion algorithm uses a spanning tree of the computer network to reduce the number of messages exchanged per critical section execution.
- The algorithm exchanges only $O(\log N)$ messages under light load, and approximately four messages under heavy load to execute the CS, where N is the number of nodes in the network.

Raymond's tree-based algorithm

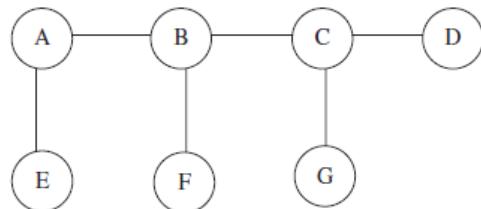
- The algorithm assumes that the underlying network guarantees message delivery.
- The time or order of message arrival cannot be predicted.
- All nodes of the network are completely reliable.
- If the network is viewed as a graph, where the nodes in the network are the vertices of the graph, and the links between nodes are the edges of the graph, a spanning tree of a network of N nodes will be a tree that contains all N nodes.
- A minimal spanning tree is one such tree with minimum cost.
- Typically, this cost function is based on the network link characteristics.
- The algorithm operates on a minimal spanning tree of the network topology or logical structure imposed on the network.

Raymond's tree-based algorithm

- The algorithm considers the network nodes to be arranged in an unrooted tree structure.
- Messages between nodes traverse along the undirected edges of the tree in the
- The tree is also a spanning tree of the seven nodes A, B, C, D, E, F, and G.
- It also turns out to be a minimal spanning tree because it is the only spanning tree of these seven nodes.
- A node needs to hold information about and communicate only to its immediate-neighboring nodes.

Raymond's tree-based algorithm

- For example, node C holds information about and communicates only to nodes B, D, and G; it does not need to know about the other nodes A, E, and F for the operation of the algorithm.



Raymond's tree-based algorithm

- Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege to signify which node has the privilege to enter the critical section.
- Only one node can be in possession of the privilege (called the privileged node) at any time, except when the privilege is in transit from one node to another in the form of a PRIVILEGE message.
- When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

Deadlock detection in distributed systems

- Deadlocks are a fundamental problem in distributed systems and deadlock detection in distributed systems has received considerable attention in the past.
- In distributed systems, a process may request resources in any order, which may not be known a priori, and a process can request a resource while holding others.
- If the allocation sequence of process resources is not controlled in such environments, deadlocks can occur.
- A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set.

Deadlock detection in distributed systems

Deadlocks can be dealt with using any one of the following three strategies:

- deadlock prevention
- deadlock avoidance
- deadlock detection

Deadlock detection in distributed systems

- **Deadlock prevention** is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-empting a process that holds the needed resource.
- In the **deadlock avoidance** approach to distributed systems, a resource is granted to a process if the resulting global system is safe.
- **Deadlock detection** requires an examination of the status of the process–resources interaction for the presence of a deadlock condition.
- To resolve the deadlock, we have to abort a deadlocked process.

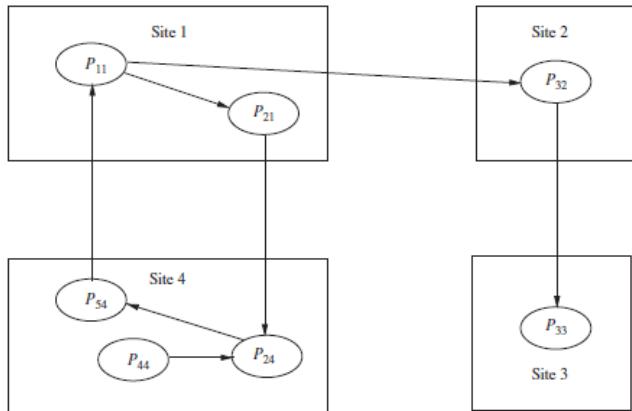
Wait-for graph (WFG)

- In distributed systems, the state of the system can be modeled by directed graph, called a wait-for graph (WFG).
- In a WFG, nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Wait-for graph (WFG)

- A WFG, where process P11 of site 1 has an edge to process P21 of site 1 and an edge to process P32 of site 2.
- Process P32 of site 2 is waiting for a resource that is currently held by process P33 of site 3.
- At the same time process P21 at site 1 is waiting on process P24 at site 4 to release a resource, and so on.
- If P33 starts waiting on process P24, then processes in the WFG are involved in a deadlock depending upon the request model.

Wait-for graph (WFG)



Models of deadlocks

- Distributed systems allow many kinds of resource requests.
- A process might require a single resource or a combination of resources for its execution.
- This section introduces a hierarchy of request models starting with very restricted forms to the ones with no restrictions whatsoever.
- This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

Models of deadlocks

The single-resource model

- The single-resource model is the simplest resource model in a distributed system, where a process can have at most one outstanding request for only one unit of a resource.
- Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

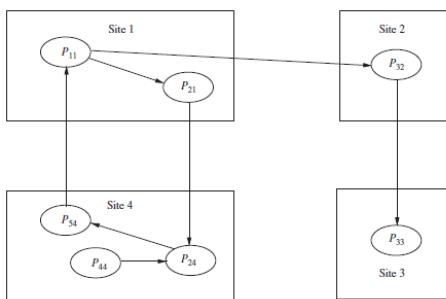
Models of deadlocks

The AND model

- In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
- The requested resources may exist at different locations.
- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Each node of the WFG in such a model is called an AND node.

Models of deadlocks

- Process P11 has two outstanding resource requests.
- In case of the AND model, P11 shall become active from idle state only after both the resources are granted.
- There is a cycle $P_{11} \rightarrow P_{21} \rightarrow P_{24} \rightarrow P_{54} \rightarrow P_{11}$, which corresponds to a deadlock situation.



Models of deadlocks

- In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa.
- That is, a process may not be a part of a cycle, it can still be deadlocked.
- Consider process P44, it is not a part of any cycle but is still deadlocked as it is dependent on P24, which is deadlocked.
- Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

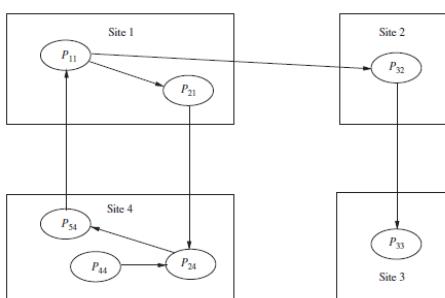
Models of deadlocks

The OR model

- In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- The requested resources may exist at different locations.
- If all requests in the WFG are OR requests, then the nodes are called OR nodes.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.

Models of deadlocks

- If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied.
- After P32 finishes execution and releases its resources, process P11 can continue with its processing.



Models of deadlocks

The AND-OR model

- A generalization of the previous two models (OR model and AND model) is the AND-OR model.
- In the AND-OR model, a request may specify any combination of and and or in the resource request.
- For example, in the AND-OR model, a request for multiple resources can be of the form x and (y or z).
- The requested resources may exist at different locations.
- To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.

Models of deadlocks

- Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.
- However, this is a very inefficient strategy.

Models of deadlocks

The (p/q) model

- Another form of the AND-OR model is the (p/q) model(called the P-out-of-Q model), which allows a request to obtain any k available resources from a pool of n resources.
- Both the models are the same in expressive power.
- However, this model lends itself to a much more compact formation of a request.
- Every request in the model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as (p/p) and OR requests for p resources can be stated as $(p/1)$

Models of deadlocks

Unrestricted model

- In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests.
- In this model, only one assumption that the deadlock is stable is made and hence it is the most general model.
- This way of looking at the deadlock problem helps in separation of concerns: concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).

Models of deadlocks

- Hence, these algorithms can be used to detect other stable properties as they deal with this general model.
- But, these algorithms are of more theoretical value for distributed systems since no further assumptions are made about the underlying distributed systems computations which leads to a great deal of overhead

Chandy–Misra–Haas algorithm for the AND model

- Chandy–Misra–Haas's distributed deadlock detection algorithm for the AND model which is based on edge-chasing.
- The algorithm uses a special message called probe, which is a triplet (i, j, k) , denoting that it belongs to a deadlock detection initiated for process P_i and it is being sent by the home site of process P_j to the home site of process P_k .
- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.

Chandy–Misra–Haas algorithm for the AND model

- A process P_j is said to be **dependent** on another process P_k if there exists a sequence of processes $P_j, P_{i1}, P_{i2}, \dots, P_{im}, P_k$ such that each process except P_k in the sequence is blocked and each process, except the P_j , holds a resource for which the previous process in the sequence is waiting.
- Process P_j is said to be locally dependent upon process P_k if P_j is dependent upon P_k and both the processes are on the same site.

Chandy–Misra–Haas algorithm for the OR model

- Chandy–Misra–Haas's distributed deadlock detection algorithm for the OR model which is based on the approach of diffusion computation
- A blocked process determines if it is deadlocked by initiating a diffusion computation.
- Two types of messages are used in a diffusion computation: $\text{query}(i, j, k)$ and $\text{reply}(i, j, k)$, denoting that they belong to a diffusion computation initiated by a process P_i and are being sent from process P_j to process P_k .

Chandy–Misra–Haas algorithm for the OR model

- A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message).
- If an active process receives a query or reply message, it discards it.
- When a blocked process P_k receives a query (i, j, k) message, it takes the following actions

Chandy–Misra–Haas algorithm for the OR model

1. If this is the first query message received by P_k for the deadlock detection initiated by P_i (called the engaging query), then it propagates the query to all the processes in its dependent set and sets a local variable $numki$ to the number of query messages sent.
2. If this is not the engaging query, then P_k returns a reply message to it immediately provided P_k has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

Chandy–Misra–Haas algorithm for the OR model

- Process P_k maintains a boolean variable wait_{ki} that denotes the fact that it has been continuously blocked since it received the last engaging query from process P_i .
- When a blocked process P_k receives a reply (i, j, k) message, it decrements num_{ki} only if wait_{ki} holds.
- A process sends a reply message in response to an engaging query only after it has received a reply to every query message it has sent out for this engaging query.
- The initiator process detects a deadlock when it has received reply messages to all the query messages it has sent out.

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

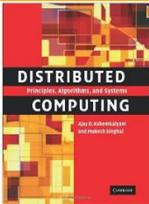
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



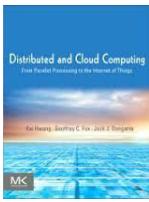
CS-9 : Consensus & Agreement Algorithms

[T1: Chap – 14]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

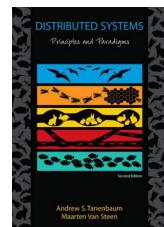


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 7

Distributed Fault Tolerance: Consensus and Agreement Algorithms, Failure Recovery

- When multiple entities cooperate with each other in solving a complex function or task in a distributed system, there are instances where a majority of these entities must agree on certain decisions without which the task cannot be solved.
- This module will discuss ways to achieve consensus and failure recovery. The module will also discuss checkpointing and logging techniques including lineage models for fault recovery.

Contact Session – 9

Distributed Fault Tolerance: Consensus and Agreement Algorithms

Problem definition

- The Byzantine agreement and other consensus problems
- Overview of Results
- Agreement in failure-free system (synchronous or asynchronous)
- Agreement in (message-passing) synchronous systems with failures

Presentation Overview

- Consensus
- Byzantine Behavior
- Agreement in a failure-free system
- Agreement In (Message-passing) Synchronous Systems With Failures
 - Consensus algorithm for crash failures (synchronous system)
 - Consensus algorithms for Byzantine failures
 - Byzantine Agreement Tree Algorithm
- Agreement In Asynchronous Message-passing Systems With Failures
 - Impossibility result for the consensus problem
 - Terminating reliable broadcast
 - Distributed transaction commit
 - k-set consensus
 - Approximate agreement
 - Renaming problem
 - Reliable broadcast

Consensus and Agreement Algorithms

- Agreement among the processes in a distributed system is a fundamental requirement for a wide range of applications.
- Many forms of coordination require the processes to exchange information to negotiate with one another and eventually reach a common understanding or agreement, before taking application-specific actions.
- A classical example is that of the commit decision in database systems, wherein the processes collectively decide whether to commit or abort a transaction that they participate in.

Consensus and Agreement Algorithms

- Consensus decision-making is a group decision-making process in which group members develop, and agree to support, a decision in the best interest of the whole.
- Consensus may be defined professionally as an acceptable resolution, one that can be supported, even if not the "favourite" of each individual.

Byzantine Behavior

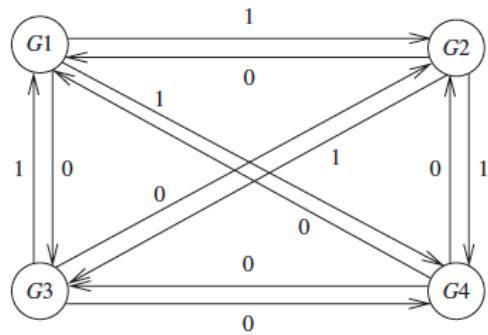
- Consider the difficulty of reaching agreement using the following example, that is inspired by the long wars fought by the Byzantine Empire in the Middle Ages.
- Four camps of the attacking army, each commanded by a general, are camped around the fort of Byzantium.
- They can succeed in attacking only if they attack simultaneously. Hence, they need to reach agreement on the time of attack.
- The only way they can communicate is to send messengers among themselves.

Byzantine Behavior

- The messengers model the messages.
- An asynchronous system is modeled by messengers taking an unbounded time to travel between two camps.
- A lost message is modeled by a messenger being captured by the enemy.
- A Byzantine process is modeled by a general being a traitor.
- The traitor will attempt to subvert the agreement-reaching mechanism, by giving misleading information to the other generals.

Byzantine Behavior

Byzantine generals sending confusing messages.



Byzantine Behavior

- For example, a traitor may inform one general to attack at 10 a.m., and inform the other generals to attack at noon.
- Or he may not send a message at all to some general. Likewise, he may tamper with the messages he gets from other generals, before relaying those messages.
- Four generals are shown, and a consensus decision is to be reached about a boolean value.
- The various generals are conveying potentially misleading values of the decision variable to the other generals, which results in confusion.

Byzantine Behavior

- In the face of such Byzantine behavior, the challenge is to determine whether it is possible to reach agreement, and if so under what conditions.
- If agreement is reachable, then protocols to reach it need to be devised.

Agreement in a failure-free system

- In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a “decision,” and distributing this decision in the system.
- A distributed mechanism would have each process broadcast its values to others, and each process computes the same function on the values received.
- The decision can be reached by using an application specific function
- Some simple examples being the majority , max , and min functions.

Agreement In (Message-passing) Synchronous Systems With Failures

Consensus algorithm for crash failures (synchronous system)

- Consensus algorithm for n processes, where up to f processes, may fail in the fail-stop model .
- Here, the consensus variable x is integer-valued. Each process has an initial value x_i .
- If up to f failures are to be tolerated, then the algorithm has $f + 1$ rounds.
- In each round, a process i sends the value of its variable x_i to all other processes if that value has not been sent before.
- Of all the values received within the round and its own value x_i at the start of the round, the process takes the minimum, and updates x_i .
- After $f + 1$ rounds, the local value x_i is guaranteed to be the consensus value.

Consensus algorithm for crash failures (synchronous system)

- A lower bound on the number of rounds
- At least $f + 1$ rounds are required, where $f < n$.
- The idea behind this lower bound is that in the worst-case scenario, one process may fail in each round; with $f + 1$ rounds, there is at least one round in which no process fails.
- In that guaranteed failure-free round, all messages broadcast can be delivered reliably, and all processes that have not failed can compute the common function of the received values to reach an agreement value.

Consensus algorithm for crash failures (synchronous system)

Upper bound on Byzantine processes

In a system of n processes, the Byzantine agreement problem (as also the other variants of the agreement problem) can be solved in a synchronous system only if the number of Byzantine processes f is such that

$$f \leq \lfloor \frac{n-1}{3} \rfloor$$

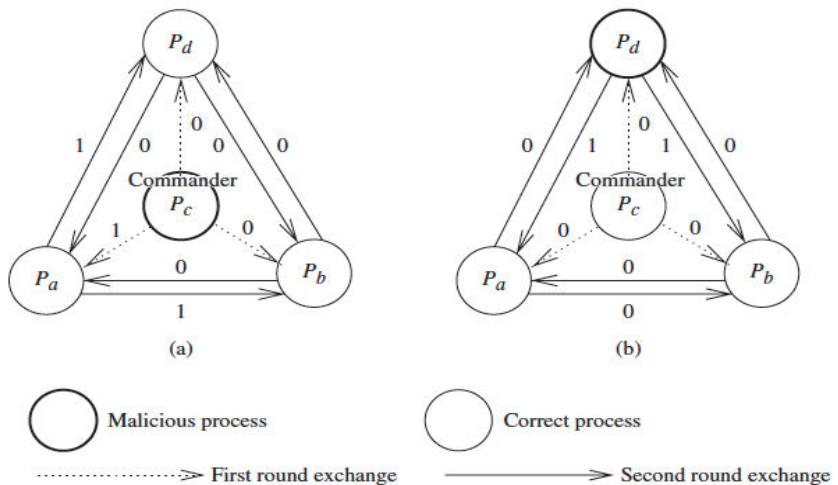
Byzantine Agreement Tree Algorithm

- We begin with an informal description of how agreement can be achieved with $n = 4$ and $f = 1$ processes
- In the first round, the commander P_c sends its value to the other three lieutenants, as shown by dotted arrows.
- In the second round, each lieutenant relays to the other two lieutenants, the value it received from the commander in the first round.
- At the end of the second round, a lieutenant takes the majority of the values it received (i) directly from the commander in the first round, and (ii) from the other two lieutenants in the second round.

Byzantine Agreement Tree Algorithm

- The majority gives a correct estimate of the “commander’s” value. Consider Figure where the commander is a traitor.
- The values that get transmitted in the two rounds are as shown.
- All three lieutenants take the majority of $(1, 0, 0)$ which is “0,” the agreement value.
- In Figure lieutenant P_d is malicious. Despite its behavior as shown, lieutenants P_a and P_b agree on “0,” the value of the commander.

Byzantine Agreement Tree Algorithm



Agreement In Asynchronous Message-passing Systems With Failures

Impossibility result for the consensus problem

- Fischer et al showed a fundamental result on the impossibility of reaching agreement in an asynchronous (message-passing) system, even if a single process is allowed to have a crash failure.
- This result has a significant impact on the field of designing distributed algorithms in a failure-susceptible system.
- The correctness proof of this result also introduced the important notion of valency of global states.

Terminating reliable broadcast

- Consider the terminating reliable broadcast problem, which states that a correct process always gets a message even if the sender crashes while sending. If the sender crashes while sending the message, the message may be a null message but it must be delivered to each correct process.
- We have an additional termination condition, which states that each correct process must eventually deliver some message.

Terminating reliable broadcast

- **Validity** If the sender of a broadcast message m is non-faulty, then all correct processes eventually deliver m .
- **Agreement** If a correct process delivers a message m , then all correct processes deliver m .
- **Integrity** Each correct process delivers a message at most once. Further, if it delivers a message different from the null message, then the sender must have broadcast m .
- **Termination** Every correct process eventually delivers some message.

Terminating reliable broadcast

- A process decides on a “0” or “1” depending on whether it receives “0” or “1” in the message from this process.
- However, if it receives the null message, it decides on a default value.
- As the broadcast is done using the terminating reliable broadcast, it can be seen that the conditions of the consensus problem are satisfied.
- But as consensus is not solvable, an algorithm to implement terminating reliable broadcast cannot exist.

Distributed transaction commit

- Database transactions require the commit operation to preserve the ACID properties (atomicity, consistency, integrity, durability) of transactional semantics.
- The commit operation requires polling all participants whether the transaction should be committed or rolled back.
- Even a single rollback vote requires the transaction to be rolled back.
- Whatever the decision, it is conveyed to all the participants in the transaction.

Distributed transaction commit

- Despite the unsolvability of the distributed commit problem under crash failure, the (blocking) two-phase commit and the non-blocking three-phase commit protocols do solve the problem.
- This is because the protocols use a somewhat different model in practice, than that used for our theoretical analysis of the consensus problem.
- The two-phase protocol waits indefinitely for a reply, and it is assumed that a crashed node eventually recovers and sends in its vote.

Distributed transaction commit

- Optimizations such as presumed abort and presumed commit are pessimistic and optimistic solutions that are not guaranteed to be correct under all circumstances.
- Similarly, the three-phase commit protocol uses timeouts to default to the “abort” decision when the coordinator does not get a reply from all the participants within the timeout period.

k-set consensus

- Although consensus is not solvable in an asynchronous system under crash failures, a weaker version, known as the k -set consensus problem, is solvable as long as the number of crash failures f is less than the parameter k .
- The parameter k indicates that the non faulty processes agree on different values, as long as the size of the set of values agreed upon is bounded by k .

k-set consensus

- The k -agreement condition is new, the validity condition is different from that for regular consensus, and the termination condition is unchanged from that for regular consensus.
- The protocol in Algorithm 14.5 can be seen to solve k -set consensus in a straightforward manner, as long as the number of crash failures f is less than k .
- Let $n = 10$, $f = 2$, $k = 3$ and let each process propose a unique value from $[1, 2, \dots, 10]$. Then the 3 -set is $[8, 9, 10]$.

Approximate agreement

- Another weaker version of consensus that is solvable in an asynchronous system under crash failures is known as the approximate consensus problem.
- Like k -set consensus, approximate agreement also assumes the consensus value is from a multi-valued domain.
- However, rather than restricting the set of consensus values to a set of size k , approximate agreement requires that the agreed upon values by the non-faulty processes be within of each other.

Renaming problem

- The consensus problem which was a problem about agreement required the processes to agree on a single value, or a small set of values (k -set consensus), or a set of values close to one another (approximate agreement), or reach agreement with high probability (probabilistic or randomized agreement).
- A different agreement problem introduced by Attiya et al. requires the processes to agree on necessarily distinct values.
- This problem is termed as the renaming problem.
- The renaming problem assigns to each process P_i , a name m_i from a domain M

Renaming problem

- The renaming problem is useful for name space transformation.
- A specific example where this problem arises is when processes from different domains need to collaborate, but must first assign themselves distinct names from a small domain.
- A second example of the use of renaming is when processes need to use their names as “tags” to simply mark their presence, as in a priority queue.

Reliable broadcast

- Although reliable terminating broadcast (RTB) is not solvable under failures a weaker version of RTB, namely reliable broadcast, in which the termination condition is dropped, is solvable under crash failures.
- The key difference between RTB and reliable broadcast is that RTB requires eventual delivery of some message – even if the sender fails just when about to broadcast.
- In this case, a null message must get sent, whereas this null message need not be sent under reliable broadcast.

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

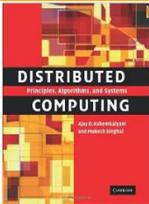
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



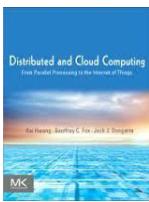
CS-10 : Distributed Fault Tolerance: Checkpointing and Rollback.

[T1: Chap – 13]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

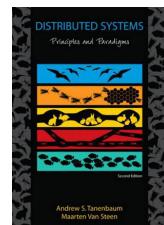


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 7

Distributed Fault Tolerance: Consensus and Agreement Algorithms, Failure Recovery

- When multiple entities cooperate with each other in solving a complex function or task in a distributed system, there are instances where a majority of these entities must agree on certain decisions without which the task cannot be solved.
- This module will discuss ways to achieve consensus and failure recovery. The module will also discuss checkpointing and logging techniques including lineage models for fault recovery.

Contact Session – 10

Distributed Fault Tolerance:

- Checkpointing and Rollback.
- Checkpointing algorithms.
- Log-based Rollback recoveries.
- Lineage-based models

Presentation Overview

- Introduction
- Checkpoint-based recovery
 - Uncoordinated checkpointing
 - Coordinated checkpointing
 - Blocking Coordinated checkpointing
 - Non - Blocking Coordinated checkpointing
 - Communication-induced checkpointing
- Log-based rollback recovery
 - Deterministic and non-deterministic events
 - Pessimistic logging
 - Optimistic logging
 - Causal logging

Introduction

- Distributed systems are not fault-tolerant and the vast computing potential of these systems is often hampered by their susceptibility to failures.
- Many techniques have been developed to add reliability and high availability to distributed systems.
- These techniques include transactions, group communication, and rollback recovery.
- These techniques have different tradeoffs and focus.
- Rollback recovery protocols restore the system back to a consistent state after a failure.

Introduction

- Rollback recovery treats a distributed system application as a collection of processes that communicate over a network.
- It achieves fault tolerance by periodically saving the state of a process during the failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work.
- The saved state is called a **checkpoint**, and the procedure of restarting from a previously checkpointed state is called **rollback recovery**.
- A checkpoint can be saved on either the stable storage or the volatile storage depending on the failure scenarios to be tolerated.

Introduction

- In distributed systems, rollback recovery is complicated because messages induce inter-process dependencies during failure-free operation.
- Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called a **rollback propagation**.
- This phenomenon of cascaded rollback is called the **domino effect**.
- Rollback propagation may extend back to the initial state of the computation, losing all the work performed before the failure.

Introduction

- In a distributed system, if each participating process takes its checkpoints independently, then the system is susceptible to the domino effect.
- This approach is called **independent or uncoordinated checkpointing**.
- It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it.
- One such technique is **coordinated checkpointing** where processes coordinate their checkpoints to form a system-wide consistent state.
- In case of a process failure, the system state can be restored to such a consistent set of checkpoints, preventing the rollback propagation.

Introduction

- Alternatively, **communication-induced checkpointing** forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes.
- Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

Introduction

- **Log-based rollback recovery** combines checkpointing with logging of nondeterministic events.
- Log-based rollback recovery relies on the piecewise deterministic (PWD) assumption, which postulates that all non-deterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's determinant.
- By logging and replaying the non-deterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed.

Introduction

- Log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints.
- It is therefore particularly attractive for applications that frequently interact with the outside world, which consists of input and output devices that cannot roll back.

Checkpoint-based recovery

- In the checkpoint-based recovery approach, the state of each process and the communication channel is checkpointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints.
- Checkpoint-based protocols are less restrictive and simpler to implement than log-based rollback recovery.
- However, checkpoint-based rollback recovery does not guarantee that prefailure execution can be deterministically regenerated after a rollback.
- Therefore, checkpoint-based rollback recovery may not be suitable for applications that require frequent interactions with the outside world.

Checkpoint-based recovery

Checkpoint-based rollback-recovery techniques can be classified into three categories:

1. Uncoordinated checkpointing
2. Coordinated checkpointing
3. Communication-induced checkpointing

Uncoordinated checkpointing

- In uncoordinated checkpointing, each process has autonomy in deciding when to take checkpoints.
- This eliminates the synchronization overhead as there is no need for coordination between processes and it allows processes to take checkpoints when it is most convenient or efficient.
- The main advantage is the lower runtime overhead during normal execution, because no coordination among processes is necessary.
- Autonomy in taking checkpoints also allows each process to select appropriate checkpoints positions.

Uncoordinated checkpointing

- However, uncoordinated checkpointing has several shortcomings
- First, there is the possibility of the domino effect during a recovery, which may cause the loss of a large amount of useful work.
- Second, recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints.
- Since no coordination is done at the time the checkpoint is taken, checkpoints taken by a process may be useless checkpoints.
- Useless checkpoints are undesirable because they incur overhead and do not contribute to advancing the recovery line.

Uncoordinated checkpointing

- Third, uncoordinated checkpointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer required.
- Fourth, it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

Coordinated checkpointing

- In coordinated checkpointing, processes orchestrate their checkpointing activities
- so that all local checkpoints form a consistent global state.
- Coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint.
- Also, coordinated checkpointing requires each process to maintain only one checkpoint on the stable storage, reducing the storage overhead and eliminating the need for garbage collection.
- The main disadvantage of this method is that large latency is involved in committing output, as a global checkpoint is needed before a message is sent.
- Also, delays and overhead are involved every time a new global checkpoint is taken.

Coordinated checkpointing

- If perfectly synchronized clocks were available at processes, the following simple method can be used for checkpointing: all processes agree at what instants of time they will take checkpoints, and the clocks at processes trigger the local checkpointing actions at all processes.
- Since perfectly synchronized clocks are not available, the following approaches are used to guarantee checkpoint consistency: either the sending of messages is blocked for the duration of the protocol, or checkpoint indices are piggybacked to avoid blocking.

Blocking Coordinated checkpointing

- A straightforward approach to coordinated checkpointing is to block communications while the check pointing protocol executes.
- After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete.
- The coordinator takes a checkpoint and broadcasts a request message to all processes, asking them to take a checkpoint.
- When a process receives this message, it stops its execution, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgment message back to the coordinator.

Blocking Coordinated checkpointing

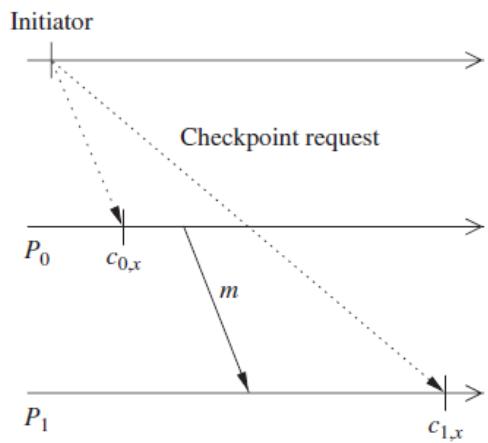
- After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase checkpointing protocol.
- After receiving the commit message, a process removes the old permanent checkpoint and atomically makes the tentative checkpoint permanent and then resumes its execution and exchange of messages with other processes.
- A problem with this approach is that the computation is blocked during the checkpointing and therefore, non-blocking checkpointing schemes are preferable.

Non - Blocking Coordinated checkpointing

- In this approach the processes need not stop their execution while taking checkpoints.
- A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

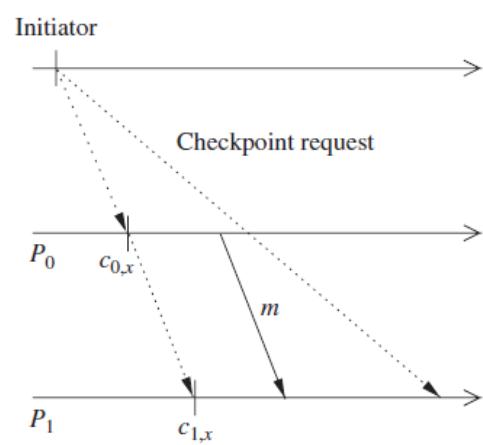
Non - Blocking Coordinated checkpointing

- Message m is sent by P_0 after receiving a checkpoint request from the checkpoint coordinator.
- Assume m reaches P_1 before the checkpoint request.
- This situation results in an inconsistent checkpoint since checkpoint $c_{1,x}$ shows the receipt of message m from P_0 , while checkpoint $c_{0,x}$ does not show m being sent from P_0 .



Non - Blocking Coordinated checkpointing

- If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message.
- An example of a non-blocking checkpoint coordination protocol using this idea is the snapshot algorithm of Chandy and Lamport in which markers play the role of the check point request messages.



Non - Blocking Coordinated checkpointing

- If the channels are non-FIFO, the following two approaches can be used: First, the marker can be piggybacked on every post-checkpoint message.
- When a process receives an application message with a marker, it treats it as if it has received a marker message, followed by the application message.
- Alternatively, checkpoint indices can serve the same role as markers, where a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked checkpoint index.

Communication-induced checkpointing

- Communication-induced checkpointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently.
- Processes may be forced to take additional checkpoints (over and above their autonomous checkpoints), and thus process independence is constrained to guarantee the eventual progress of the recovery line.
- Communication-induced checkpointing reduces or completely eliminates the useless checkpoints.

Communication-induced checkpointing

- In communication-induced checkpointing, processes take two types of checkpoints, namely, autonomous and forced checkpoints.
- The checkpoints that a process takes independently are called **local checkpoints**, while those that a process is forced to take are called **forced checkpoints**.

Communication-induced checkpointing

- Communication-induced checkpointing piggybacks protocol-related information on each application message.
- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line.
- The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring some latency and overhead.
- It is therefore desirable in these systems to minimize the number of forced checkpoints.
- In contrast with coordinated checkpointing, no special coordination messages are exchanged.

Communication-induced checkpointing

There are two types of communication-induced checkpointing :

1. Model based checkpointing
2. Index-based checkpointing

- In model-based checkpointing, the system maintains checkpoints and communication structures that prevent the domino effect or achieve some even stronger properties.
- In index-based checkpointing, the system uses an indexing scheme for the local and forced checkpoints, such that the checkpoints of the same index at all processes form a consistent state.

Communication-induced checkpointing

Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints.
- A process detects the potential for inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns.
- A forced checkpoint is generally used to prevent the undesirable patterns from occurring.
- No control messages are exchanged among the processes during normal operation.
- All information necessary to execute the protocol is piggybacked on application messages.
- The decision to take a forced checkpoint is done locally using the information available.

Communication-induced checkpointing

Index-based checkpointing

- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.
- Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint.

Log-based rollback recovery

- A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.

Deterministic and non-deterministic events

- Log-based rollback recovery exploits the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a non-deterministic event.
- A non-deterministic event can be the receipt of a message from another process or an event internal to the process.
- Note that a message send event is not a non-deterministic event.

Deterministic and non-deterministic events

- Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage.
- During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage.
- Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery.
- After a failure occurs, the failed processes recover by using the checkpoints and logged determinants to replay the corresponding non-deterministic events precisely as they occurred during the pre-failure execution.

Pessimistic logging

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation.
- This assumption is “pessimistic” since in reality failures are rare.
- In their most straightforward form, pessimistic protocols log to the stable storage the determinant of each non-deterministic event before the event affects the computation.

Optimistic logging

- In optimistic logging protocols, processes log determinants asynchronously to the stable storage.
- These protocols optimistically assume that logging will be complete before a failure occurs.
- Determinants are kept in a volatile log, and are periodically flushed to the stable storage.
- Thus, optimistic logging does not require the application to block waiting for the determinants to be written to the stable storage, and therefore incurs much less overhead during failure-free execution.
- However, the price paid is more complicated recovery, garbage collection, and slower output commit.

Causal logging

- Causal logging combines the advantages of both pessimistic and optimistic logging at the expense of a more complex recovery protocol.
- Like optimistic logging, it does not require synchronous access to the stable storage except during output commit.
- Like pessimistic logging, it allows each process to commit output independently and never creates orphans, thus isolating processes from the effects of failures at other processes.
- Moreover, causal logging limits the rollback of any failed process to the most recent checkpoint on the stable storage, thus minimizing the storage overhead and the amount of lost work.

THANK YOU

IMP Note to Self



STOP RECORDING



CC ZG526

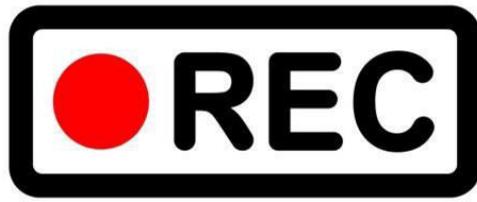
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

BITS Pilani
Pilani | Dubai | Goa | Hyderabad



IMP Note to Self



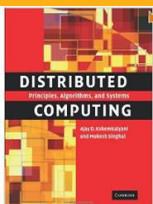
START RECORDING

CC ZG526 Distributed Computing || Dt: 20th April 2025 2 BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

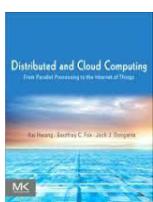
CS-11 : Peer-to-Peer computing and Overlay graphs

[T1: Chap – 18 & R1 : Chap - 14]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

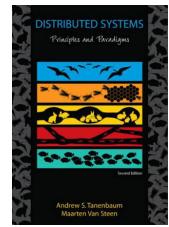


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 8

Peer-to-Peer computing and Overlay graphs

- P2P architecture is used for build various large scale-out systems, e.g. databases, caching, content distribution, digital currency etc.
- This module covers the architectural and algorithm design options for a P2P overlay network.

Contact Session – 11

Peer-to-Peer computing and Overlay graphs

- Introduction
- Data indexing and Overlays
- Unstructured Overlays
- Structured Overlays: CHORD DHT

Presentation Overview

- Introduction
- Characteristics and performance features of P2P systems
- Data indexing and overlays
 - Centralized indexing
 - Local indexing
 - Distributed indexing
 - Structured overlays
 - Unstructured overlays
- Challenges in P2P system design

Introduction

- Peer-to-peer (P2P) network systems use an application-level organization of the network overlay for flexibly sharing resources (e.g., files and multimedia documents) stored across network-wide computers.
- In contrast to the client–server model, any node in a P2P network can act as a server to others and, at the same time, act as a client.

Introduction

- Communication and exchange of information is performed directly between the participating peers and the relationships between the nodes in the network are equal.
- Thus, P2P networks differ from other Internet applications in that they tend to share data from a large number of end users rather than from the more central machines and Web servers.

Introduction

Several well-known P2P networks that allow P2P file-sharing include

- Napster
- Gnutella
- Freenet
- Pastry
- Chord
- CAN

Introduction

- Traditional distributed systems used DNS (domain name service) to provide a lookup from host names (logical names) to IP addresses.
- Special DNS servers are required, and manual configuration of the routing information is necessary to allow requesting client nodes to navigate the DNS hierarchy.
- Further, DNS is confined to locating hosts or services (not data objects that have to be a priori associated with specific computers), and host names need to be structured as per administrative boundary regulations.
- P2P networks overcome these drawbacks, and, more importantly, allow the location of arbitrary data objects.

Introduction

- An important characteristic of P2P networks is their ability to provide a large combined storage, CPU power, and other resources while imposing a low cost for scalability, and for entry into and exit from the network.
- The ongoing entry and exit of various nodes, as well as dynamic insertion and deletion of objects is termed as *churn*.
- The impact of churn should be as transparent as possible.
- P2P networks exhibit a high level of self-organization and are able to operate efficiently despite the lack of any prior infrastructure or authority.

Characteristics and performance features of P2P systems

Features	Performance
Self-organizing	Large combined storage, CPU power, and resources
Distributed control	Fast search for machines and data objects
Role symmetry for nodes	Scalable
Anonymity	Efficient management of churn
Naming mechanism	Selection of geographically close servers
Security, authentication, trust	Redundancy in storage and paths

Napster

- One of the earliest popular P2P systems, Napster, used a server-mediated central index architecture organized around clusters of servers that store direct indices of the files in the system.
- The central server maintains a table with the following information of each registered client:
 - (i) the client's address (IP) and port, and offered bandwidth
 - (ii) information about the files that the client can allow to share.

Napster

The basic steps of operation to search for content and to determine a node from which to download the content are the following:

1. A client connects to a meta-server that assigns a lightly loaded server from one of the close-by clusters of servers to process the client's query.
2. The client connects to the assigned server and forwards its query along with its own identity.
3. The server responds to the client with information about the users connected to it and the files they are sharing.
4. On receiving the response from the server, the client chooses one of the users from whom to download a desired file. The address to enable the P2P connection between the client and the selected user is provided by the server to the client.

Napster

- Users are generally anonymous to each other.
- The directory serves to provide the mapping from a particular host that contains the required content, to the IP address needed to download from it.

Data indexing and overlays

- The data in a P2P network is identified by using indexing.
- Data indexing allows the physical data independence from the applications.
- Indexing mechanisms can be classified as being
 - Centralized
 - Local
 - Distributed

Centralized indexing

- Centralized indexing entails the use of one or a few central servers to store references (indexes) to the data on many peers.
- The DNS lookup as well as the lookup by some early P2P networks such as Napster used a central directory lookup.

Local indexing

- Local indexing requires each peer to index only the local data objects and remote objects need to be searched for.
- This form of indexing is typically used in unstructured overlays

Distributed indexing

- Distributed indexing involves the indexes to the objects at various peers being scattered across other peers throughout the P2P network.
- In order to access the indexes, a structure is used in the P2P overlay to access the indexes.
- Distributed indexing is the most challenging of the indexing schemes, and many novel mechanisms have been proposed, most notably the *distributed hash table (DHT)*.
- Various DHT schemes differ in the hash mapping, search algorithms, diameter for lookup, search diameter, fault-tolerance, and resilience to churn.

Structured overlays

- The P2P network topology has a definite structure, and the placement of files or data in this network is highly deterministic as per some algorithmic mapping.
- The objective of such a deterministic mapping is to allow a very fast and deterministic lookup to satisfy queries for the data.
- These systems are termed as *lookup systems* and typically use a hash table interface for the mapping.
- The hash function, which efficiently maps *keys* to *values*, in conjunction with the regular structure of the overlay, allows fast search for the location of the file.

Structured overlays

- An implicit characteristic of such a deterministic mapping of a file to a location is that the mapping can be based on a single characteristic of the file such as its name, its length, or more generally some *predetermined* function computed on the file.
- A disadvantage of such a mapping is that arbitrary queries, such as range queries, attribute queries and exact keyword queries cannot be handled directly.

Unstructured overlays

- The P2P network topology does not have any particular controlled structure, nor is there any control over where files/data is placed.
- Each peer typically indexes only its local data objects, hence, *local indexing* is used.
- Node joins and departures are easy – the local overlay is simply adjusted.
- File placement is not governed by the topology.
- Search for a file may entail high message overhead and high delays.

Unstructured overlays

- Although the P2P network topology does not have any controlled structure, some topologies naturally emerge
- Unstructured overlays have the serious disadvantage that queries may take a long time to find a file or may be unsuccessful even if the queried object exists.
- The message overhead of a query search may also be high.

Unstructured overlays

The following are the main advantages of unstructured overlays

- Exact keyword queries, range queries, attribute-based queries, and other complex queries can be supported because the search query can capture the semantics of the data being sought; and the indexing of the files and data is not bound to any non-semantic structure.
- Unstructured overlays can accommodate high churn, i.e., the rapid joining and departure of many nodes without affecting performance.

Unstructured overlays

- Unstructured overlays are efficient when there is some degree of data replication in the network.
- Users are satisfied with a best-effort search.
- The network is not so large as to lead to scalability problems during the search process.

Semantic Index mechanism

- An alternate way to classify indexing mechanisms is as being a *semantic index mechanism* or a *semantic-free* index mechanism.
- A semantic index is human readable, for example, a document name, a keyword, or a database key.
- A semantic-free index is not human readable and typically corresponds to the index obtained by a hash mechanism, e.g., the DHT schemes.
- A semantic index mechanism supports keyword searches, range searches, and approximate searches, whereas these searches are not supported by semantic free index mechanisms.

Challenges in P2P system design

Fairness

- P2P systems depend on all the nodes cooperating to store objects and allowing other nodes to download from them.
- However, nodes tend to be selfish in nature; thus there is a tendency to download files without reciprocating by allowing others to download the locally available files.
- This behavior, termed as *leaching* or *free-riding*, leads to a degradation of the overall P2P system performance.
- Hence, penalties and incentives should be built in the system to encourage sharing and maximize the benefit to all nodes.

Challenges in P2P system design

- In the prisoners' dilemma, two suspects, A and B, are arrested by the police.
- There is not enough evidence for a conviction.
- The police separate the two prisoners, and, separately, offer each the same deal: if the prisoner testifies against (betrays) the other prisoner and the other prisoner remains silent, the betrayer gets freed and the silent accomplice gets a 10-year sentence.
- If both testify against the other (betray), they each receive a 2-year sentence.
- If both remain silent, the police can only sentence both to a small 6-month term on a minor offence.

Challenges in P2P system design

- Rational selfish behavior dictates that both A and B would betray the other.
- This is not a Pareto-optimal solution, where a Pareto-optimal solution is one in which the overall good of all the participants is maximized.

Challenges in P2P system design

- In the above example, both A and B staying silent results in a Pareto-optimal solution.
- The dilemma is that this is not considered the rational behavior of choice.
- In the iterative prisoners' dilemma, the game is played multiple times, until an "equilibrium" is reached.
- Each player retains memory of the last move of both players (in more general versions, the memory extends to several past moves).
- After trying out various strategies, both players should converge to the ideal optimal solution of staying silent. This is Pareto-optimal.

Challenges in P2P system design

- The commonly accepted view is that the *tit-for-tat* strategy, is the best for winning such a game.
- In the first step, a prisoner cooperates, and in each subsequent step, he reciprocates the action taken by the other party in the immediately preceding step.
- The BitTorrent P2P system has adopted the tit-for-tat strategy in deciding whether to allow a download of a file in solving the leaching problem.
- Here, cooperation is analogous to allowing others to upload local files, and betrayal is analogous to not allowing others to upload.

Challenges in P2P system design

Trust or reputation management

- Various incentive-based economic mechanisms to ensure maximum cooperation among the selfish peers inherently depend on the notion of trust.
- In a P2P environment where the peer population is highly transient, there is also a need to have trust in the quality of data being downloaded.
- These requirements have lead to the area of trust and trust management in P2P Systems
- As no node has a complete view of the other downloads in the P2P system, it may have to contact other nodes to evaluate the trust in particular offerers from which it could download some file.

Challenges in P2P system design

- These communication protocol messages for trust management may be susceptible to various forms of malicious attack (such as man-in-the-middle attacks and Sybil attacks), thereby requiring strong security guarantees.
- The many challenges to tracking trust in a distributed setting include: quantifying trust and using different metrics for trust, how to maintain trust about other peers in the face of collusion, and how to minimize the cost of the trust management protocols.

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

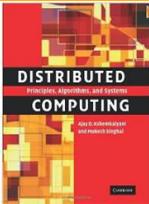


BITS Pilani
Pilani | Dubai | Goa | Hyderabad

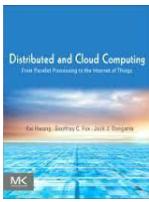
CS-12 : Peer-to-Peer computing and Overlay graphs

[T1: Chap – 18 & R1 : Chap - 14]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

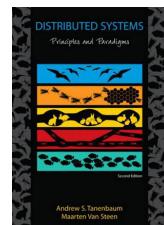


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 8

Peer-to-Peer computing and Overlay graphs

- P2P architecture is used for build various large scale-out systems, e.g. databases, caching, content distribution, digital currency etc.
- This module covers the architectural and algorithm design options for a P2P overlay network.

Contact Session – 12

Peer-to-Peer computing and Overlay graphs

- Graph structure of Complex networks
- Internet Graphs
- Generalized Random graph networks
- Small-world and Scale-free networks

Presentation Overview

- Graph structures of complex networks
- Internet graphs
- Generalized random graph networks
- Small-world networks
- Scale-free networks
- Security
- Classifications of Attacks
 - Classification by Target Functional Layer
 - Classification by Effect on Victims
 - Classification by Attacker Goals
- Security Mechanisms

Graph structures of complex networks

- P2P overlay graphs can have different structures.
- An intriguing question is to characterize the structure of overlay graphs.
- This question is a small part of a much wider challenge of how to characterize large networks that grow in a distributed manner without any coordination.
- Such networks exist in the following:

Graph structures of complex networks

- **Computer science:** the WWW graph (WWW), the Internet graph that models individual routers and interconnecting links (INTNET), and the autonomous systems (AS) graph in the Internet.
- **Social networks (SOC)**, the phonecall graph (PHON), the movie actor collaboration graph (ACT), the author collaboration graph (AUTH), and citation networks (CITE).
- **Linguistics:** the word co-occurrence graph (WORDOCC), and the word synonym graph (WORDSYN).
- **The power distribution grid (POWER).**
- **Nature:** in protein folding (PROT), where nodes are proteins and an edge represents that the two proteins bind together, and in substrate graphs for various bacteria and micro-organisms (SUBSTRATE), where nodes are substrates and edges are chemical reactions in which substrates participate.

Graph structures of complex networks

- The first logical attempt to model large networks without any known design principles is to use random graphs.
- The random graph model, also known as the Erdos–Renyi (ER) model, assumes n nodes and a link between each pair of nodes with probability p , leading to $n(n-1)p/2$ edges.
- However, the complex networks encountered in practice are not entirely random, and show some, somewhat intangible, organizational principles.
- Three ideas have received much investigative attention in recent times

Graph structures of complex networks

Small world networks

- Even in very large networks, the path length between any pair of nodes is relatively small.
- This principle of a “small world” was popularized by sociologist Stanley Milgram by the “six degrees of separation” uncovered between any two people.
- As the average distance between any pair of nodes in the ER model grows logarithmically with n , the ER graphs are small worlds.

Graph structures of complex networks

Clustering

- Social networks are characterized by cliques.
- The degree of cliques in a graph can be measured by various clustering coefficients, such as the following.
- Consider a node i having k_i out-edges. Let l_i be the actual number of edges among the k_i nearest neighbors of i .
- If these k_i nearest neighbors were in a clique, they would have $k_i(k_i-1)/2$ edges among them.
- The clustering coefficient for node i is $C_i = 2l_i/(k_i(k_i-1))$
- The network-wide clustering coefficient is the average of all C_i s, for all nodes i in the network.

Graph structures of complex networks

- The random graph model has a clustering coefficient of exactly p .
- As most real networks have a much larger clustering coefficient, this random graph model (ER) is unsatisfactory.

Graph structures of complex networks

Degree distributions

- Let $P(k)$ be the probability that a randomly selected node has k incident edges.
- In many networks – such as INTER, AS, WWW, SUBST – $P(k) \sim p^{-y}$ i.e., $P(k)$ is distributed with a powerlaw tail.
- Such networks that are free of any characteristic scale, i.e., whose degree characterization is independent of n , are called scale-free networks.

Graph structures of complex networks

- Current empirical measurements show the following properties of some commonly occurring graphs:
- **WWW** In-degree and out-degree distributions both follow power laws; it is a small world; and is a directed graph, but does show a high clustering coefficient.
- **INTNET** Degree distributions follow power law; small world; shows clustering.
- **AS** Degree distributions follow power law; small world; shows clustering.
- **ACT** Degree distributions follow power law tail; small world (similar path length as ER); shows high clustering.
- **AUTH** Degree distributions follow power law; small world; shows high clustering.
- **SUBSTRATE** In-degree and out-degree distributions both follow power laws; small world; large clustering coefficient.

Graph structures of complex networks

- **PROT** Degree distribution has a power law with exponential cutoff.
- **PHON** In-degree and out-degree distributions both follow power laws.
- **CITE** In-degree follows power law, out-degree has an exponential tail.
- **WORDOCC** Two-regime power-law degree distribution; small world; high clustering coefficient.
- **WORDSYN** Power-law degree distribution; small world; high clustering coefficient.
- **POWER** Degree distribution is exponential.

Internet graphs

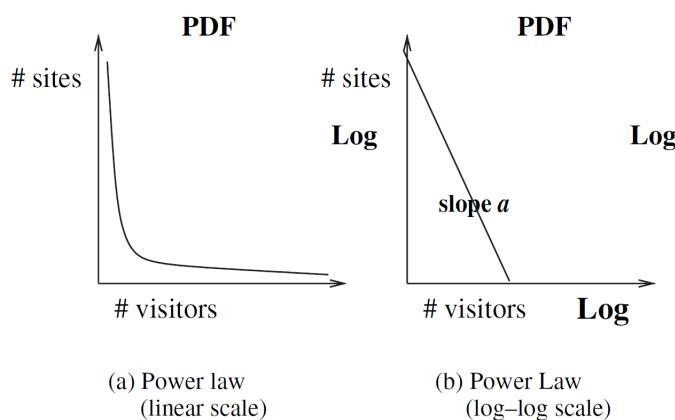
- We consider some properties of the Internet, that demonstrate a power-law behavior as measured empirically.
- The power law informally implies that large occurrences are very rare, and the frequency of the occurrence increases as the size decreases.
- Examples pertaining to the Web are: the number of links to a page, the number of pages within a Web location, and the number of accesses to a Web page.
- We begin by taking the example of the popularity of Websites to illustrate the definitions of three related observed laws: Zipf's law, the Pareto law, and the Power law:

Internet graphs

Power law

- This law is stated as a probability distribution function (PDF).
- It says that the number of occurrences of events that equal x is an inverse power of x .
- Figure 18.11(a) and (b) show the typical Power law PDF plots on both linear and log-log scales, respectively.
- In the log-log plot, the slope is a .
- In our example, this corresponds to the number of sites that have exactly x visitors.

Internet graphs

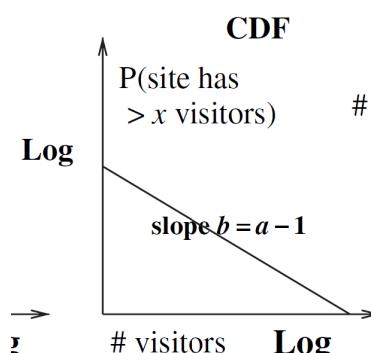


Internet graphs

Pareto law

- This law is stated as a cumulative distribution function (CDF).
- The number of occurrences larger than x is an inverse power of x .
- The CDF can be obtained by integrating the PDF.
- The exponents a and b of the Pareto (CDF) and Power laws (PDF) are related as $b+1=a$.
- Figure shows the Pareto law CDF plot on a log–log scale.
- In the log–log plot, the slope is $b = a - 1$.
- In our example, this corresponds to the number of sites that have at least x visitors.

Internet graphs



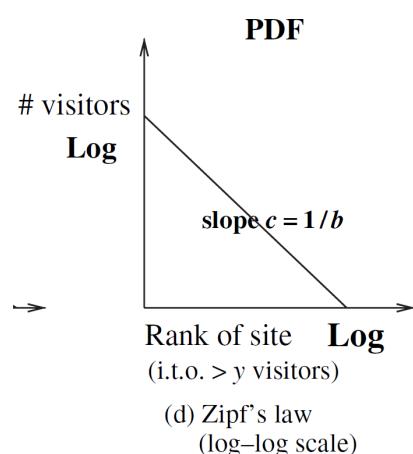
(c) Pareto law
(log–log scale)

Internet graphs

Zipf's law

- This law states the count n (i.e., the number) of the occurrences of an event, as a function of the event's rank r .
- It says that the count of the r th largest occurrence is an inverse power of the rank r .
- Figure shows the Zipf plot on a log–log scale.
- In the log–log plot, the slope is c , which, as we see below, is $1/b = 1/(a-1)$.

Internet graphs

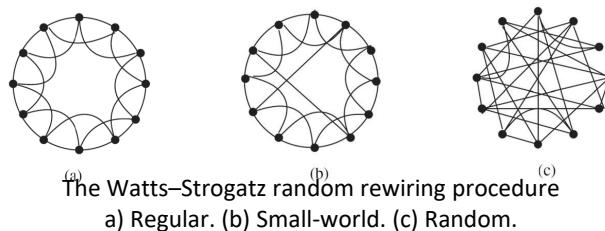


Generalized random graph networks

- Random graphs cannot capture the scale-free nature of real networks, which states that the node degree distribution follows a power law.
- The generalized random graph model uses the degree distribution as an input, but is random in all other respects.
- Thus, the constraint that the degree distribution must obey a power law is superimposed on an otherwise random selection of nodes to be connected by edges.
- These semi-random graphs can be analyzed for various properties of interest.
- Although a simple formal model for the clustering coefficient is not known, it has been observed that generalized random graphs have a random distribution of edges similar to the ER model, and hence the clustering coefficient will likely tend to zero as N increases.

Small-world networks

- Real-world networks are small worlds, having small diameter, like random graphs, but they have relatively large clustering coefficients that tend to be independent of the network size.
- Ordered lattices tend to satisfy this property that clustering coefficients are independent of the network size.



Scale-free networks

- Many real networks are scale-free, and even for those that are not scalefree, the degree distribution follows an exponential tail that is significantly different from that of the Poisson distribution.
- Semi-random graphs that are constrained to obey a power law for the degree distributions and constrained to have large clustering coefficients yield scale-free networks, but do not shed any insight into the mechanisms that give birth to scale-free networks.
- Rather than modeling the network topology, it is better to model the network assembly and evolution process.

Scale-free networks

Initially, there are m_0 isolated nodes. At each sequential step, perform one of the following operations:

Growth Add a new node with m edges, (where $m \leq m_0$), that link the new node to m different nodes already in the system.

Preferential attachment The probability \prod that the new node will be connected to node i depends on the degree k_i , such that:

$$\prod(k_i) = \frac{k_i}{\sum_j(k_j)}. \quad (18.12)$$

Scale-free networks

- Rather than begin with a constant number of nodes n that are then randomly connected or rewired, real networks (e.g., WWW, INTERNET) exhibit growth by the addition of nodes and edges.
- Rather than assume that the probability of adding (or rewiring) an edge between two nodes is a constant, real networks exhibit the property of preferential attachment, where the probability of connecting to a node depends on the node degree.

Security

- Many users wonder, “Am I leaking some private, especially financial, information when I use a P2P system?” Recent studies focusing on information leakage and inadvertent disclosures through P2P file-sharing networks found a surprising number of threats to both corporate and individual security, including a large number of searches targeted to uncover sensitive documents and data.
- In their study, 478 P2P searches and files on three P2P networks Gnutella, FastTrack, and eDonkey over a seven-week period (December 27, 2005, through February 13, 2006) were categorized.
- Sixteen thousand searches out of an estimated 800 million searches were found to be related to banking institutes.

Security

- Out of the 16,000 searches, 7,194 were found to be medium and high risk, where searches are directed for specific documents or data, such as account user information, passwords, and routing and personal identification numbers (PINs).
- These kinds of searches could fuel malicious activity and represent clear threats.
- For example, USA Today reported a case of ID theft by file sharing. The offender reportedly used Limewire's file-sharing program to troll other people's computers for financial information, which he used to open credit cards for an online shopping spree.
- At least 83 victims were identified, "Most of whom have teenage children and did not know the file-sharing software was on their computer."
- It also pointed out the possibility of the number of people affected in the order of hundreds and the total amount lost in the order of hundreds of thousands of dollars.

Security

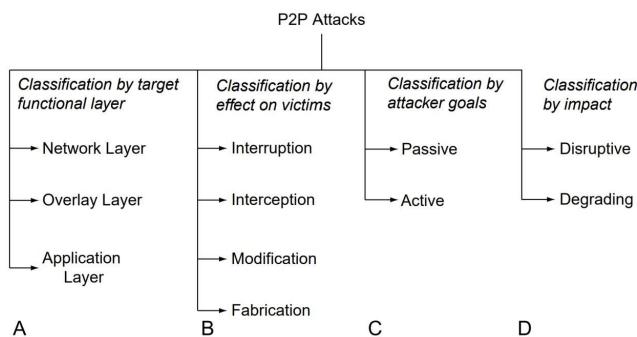
- "We believe that P2P file-sharing networks represent a significant and poorly understood threat to business, government, and individuals.
- Given the nature of the threat, we would argue that many individuals may be experiencing identity theft and fraud without ever knowing the source of their misfortune.
- Furthermore, we see many of the current P2P trends increasing the problem.
- We urge both corporate executives and government officials to educate themselves and their constituencies to the risks these networks represent," an author of these studies concluded in testimony before the Committee on Oversight and Government Reform of the United States House of Representatives.

Security

- File sharing is just one of many types of applications of P2P networks.
- The increasing popularity of P2P applications, including P2P file sharing, P2P media streaming, P2PTV, and P2P gaming, could potentially instigate security risks that are more serious than those found in and beyond the much discussed content security and copyright issues.
- It could open up opportunities for cyber criminals to trawl the P2P network and steal or gather confidential information, to damage documents, content, or even devices, and to poison the network for criminal intents.
- It is believed that much of these vulnerabilities are rooted in the autonomous and decentralized nature of P2P systems and the relatively limited use of security techniques in existing P2P applications.

Classifications of Attacks

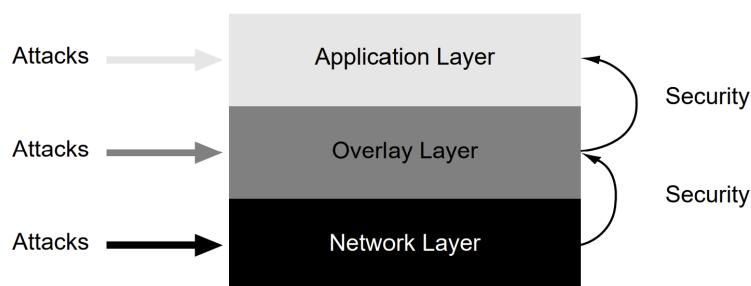
- Attacks on P2P systems can be classified based on various criteria.
- Some common criteria include functional target, communication mechanism, propagation mechanism, effect on victims, and impact.



Classification by Target Functional Layer

- Application layer security largely depends on the guarantees provided at the overlay network layer, whereas security at the overlay layer relies on the assurance offered at the network layers.
- Security breaches that take place on the application layer assume direct user-to-network interaction through application interfaces.
- Although file sharing has been the most discussed application for security concerns in P2P networks, risks and threats do exist in regard to other P2P applications as well.
- P2P file storage systems and P2P media streaming, for example, are concerned with information leakage and copyright protection as well.

Classification by Target Functional Layer



Classification by Target Functional Layer

- At the overlay layer, attacks target overlay layer operation primitives to alter or impair P2P communication.
- In structured P2P overlays, a major threat is malicious routing, whereby attackers exploit the vulnerability of the DHT routing mechanism, since peers rely on each other's routing table to function properly.
- This can be the weakest point of the overlay.
- When some or a substantial number of routing tables in the P2P overlay are compromised, the functionality of any P2P application that is built on top of the overlay may be degraded or even disrupted.

Classification by Target Functional Layer

- At the network layer, P2P overlay protocols are susceptible to existing conventional attacks that affect many other networked applications.
- These attacks include interception of packets, manipulation of packet contents, and mis-routing of packets.

Classification by Effect on Victims

As in conventional networked applications, the overlay messaging is also susceptible to four classes of attack:

- Interruption. Unauthorized disruption.
- Interception. Unauthorized access.
- Modification. Unauthorized tampering.
- Fabrication. Unauthorized creation.

Classification by Effect on Victims

- These attacks can be passive or active.
- Through exchange of information with other peers as well as through embedding attack mechanisms in the peer application software itself, a peer may easily be exposed to viruses, worms, Trojan horses, adware, or spyware. Intrusion, eavesdropping, espionage, sniffing, substitution or insertion, jamming, overload, spoofing, sabotage, spamming, reverse engineering, cryptanalysis, theft, scavenging, and denial of service are just some of the many forms of attack existing today.

Classification by Attacker Goals

- Goals of attacks in a P2P overlay, besides active intent such as theft of data, theft of resources, tampering with devices and networks, and disruption of services, also include passive intent such as traffic analysis and signal analysis.

SECURITY MECHANISMS

- Given the vulnerabilities of existing P2P overlays and the attacks we've described, can we still make a P2P overlay secure and dependable? Defending against the threats against P2P overlays requires careful planning and selection of P2P infrastructure and security mechanisms.
- Security policies are the foremost requirement in building a secure system.
- The set of rules defines and governs the control, use, and action entities of a system.
- With security policies in place, it is then possible to design suitable security mechanisms to enforce the security policies and ensure the security of the system.

SECURITY MECHANISMS

Cryptographic Solutions

- Cryptographic schemes offer the most effective solutions for many information security issues.
- They are also essential to security in P2P.
- Among various crypto tools, encryption and authentication are two fundamental and the most frequently used crypto primitives.

SECURITY MECHANISMS

DoS Countermeasures

- The most popular countermeasures of DoS attacks include service/host backup, reactive detection, rate limiting, and filtering. Having a separate emergency block of IP addresses, for example, can be invaluable in surviving a DoS attack.
- Pattern detection is often helpful by storing the signature of known attacks in a database.
- Rate-limiting mechanisms impose a rate limit on a stream that has been characterized as malicious by the detection mechanism.
- These are often used as a response technique when a detection mechanism cannot characterize the attack stream.
- Effective filtering is another way to protect against DoS and DDoS attacks.

SECURITY MECHANISMS

Secure Routing in Structured P2P

- Castro et al. define a secure routing primitive that ensures that when a nonfaulty node sends a message to a key, the message reaches all nonfaulty members in the set of replica roots with very high probability.
- Secure routing guarantees that even with the existence of malicious peers (nodes) that may corrupt, drop, modify, replace, or misroute the message, the correct message will eventually be delivered to the intended receiver with high probability

SECURITY MECHANISMS

Secure Routing Table Maintenance

- Castro et al. suggests imposing strong constraints on the set of nodelds that can fill each slot in a routing table to maintain routing table security.
- In two-table routing, one guarantees performance while the other is constrained such that the probability of it being manipulated is minimized.
- Authentication is another way to reduce routing table attacks.
- However, additional cost is expected and may reduce the performance of the P2P application, especially delay-bounded applications

SECURITY MECHANISMS

Secure Message Forwarding

- Since peer-to-peer overlays rely solely on other peers for message routing, a message has to be properly routed without modification in transit.
- An adversary may try to alter the message when routing the message to its receiver, alter its routing table to disrupt the message forwarding, or take advantage of locality to control some routes.
- Secure message forwarding ensures that at least one copy of a message sent to a peer reaches the correct peer with high probability.

IMP Note to Self



STOP RECORDING





BITS Pilani

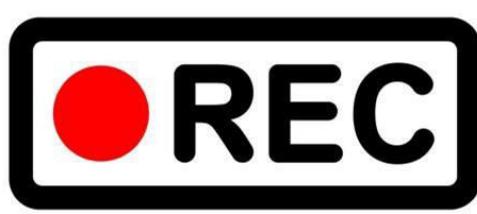
Pilani | Dubai | Goa | Hyderabad

CC ZG526

Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

CC ZG526 Distributed Computing || Dt: 27th April 2025

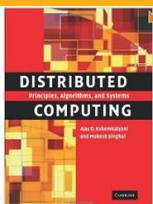
2

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

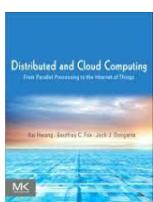
CS-13 : Cluster Computing

[R2: Chap – 2]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

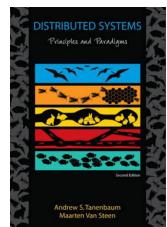


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 9

Cluster Computing

This module introduces Cluster computing as

- (i) a model of carrying out distributed computation using a collection of homogeneous computers and
- (ii) as a building block for Cloud infrastructure.

Contact Session – 13

Cluster Computing

- Cluster development trends
- Design objectives of Computer clusters
- Cluster organization and resource sharing
- Node architecture and MPP packaging

Presentation Overview

- Cluster Architecture
 - Single-System Image
- Clustering for massive parallelism
 - Cluster Development Trends
 - Milestone Cluster Systems
- Design Objectives of Computer Clusters
- Cluster Organization and Resource Sharing
 - Resource Sharing in Clusters
- Node Architectures and MPP Packaging

Cluster Architecture

- A computing cluster consists of interconnected stand-alone computers which work cooperatively as a single integrated computing resource.
- A **computer cluster** consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system.
- Unlike grid computers, computer clusters have each node set to perform the same task, controlled and scheduled by software

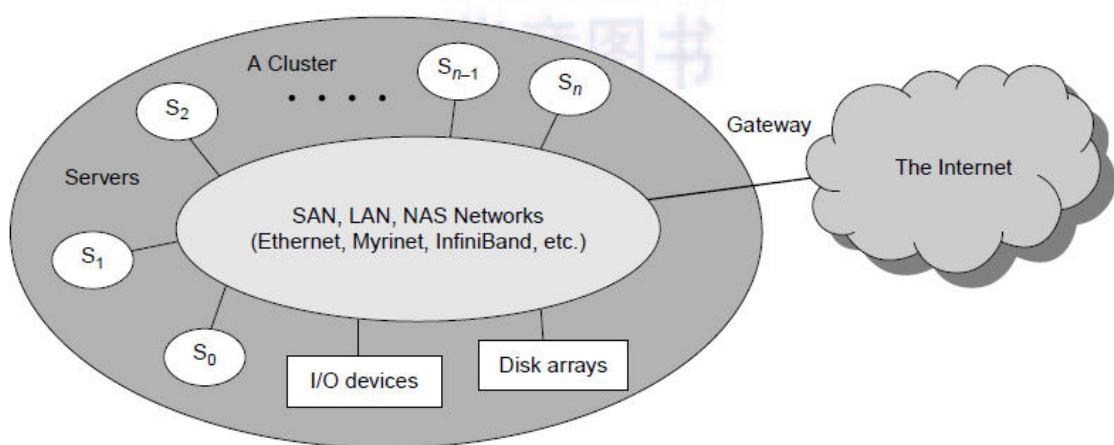
Cluster Architecture

- The architecture of a typical server cluster built around a low-latency, high bandwidth interconnection network.
- To build a larger cluster with more nodes, the interconnection network can be built with multiple levels of Gigabit Ethernet, Myrinet, or InfiniBand switches.
- Through hierarchical construction using a SAN, LAN, or WAN, one can build scalable clusters with an increasing number of nodes.
- The cluster is connected to the Internet via a virtual private network (VPN) gateway.
- The gateway IP address locates the cluster.

Cluster Architecture

- The system image of a computer is decided by the way the OS manages the shared cluster resources.
- Most clusters have loosely coupled node computers.
- All resources of a server node are managed by their own OS.
- Thus, most clusters have multiple system images as a result of having many autonomous nodes under different OS control.

Cluster Architecture



A cluster of servers interconnected by a high-bandwidth SAN or LAN with shared I/O devices and disk arrays the cluster acts as a single computer attached to the Internet

Single-System Image

- Greg Pfister has indicated that an ideal cluster should merge multiple system images into a single-system image (SSI).
- Cluster designers desire a cluster operating system or some middleware to support SSI at various levels, including the sharing of CPUs, memory, and I/O across all cluster nodes.
- An SSI is an illusion created by software or hardware that presents a collection of resources as one integrated, powerful resource.
- SSI makes the cluster appear like a single machine to the user.
- A cluster with multiple system images is nothing but a collection of independent computers.

Clustering for massive parallelism

- Clustering explores massive parallelism at the job level and achieves high availability (HA) through stand-alone operations.
- The benefits of computer clusters and massively parallel processors (MPPs) include scalable performance, HA, fault tolerance, modular growth, and use of commodity components.
- These features can sustain the generation changes experienced in hardware, software, and network components

Clustering for massive parallelism

- These features can sustain the generation changes experienced in hardware, software, and network components. Cluster computing became popular in the mid-1990s as traditional mainframes and vector supercomputers were proven to be less cost-effective in many high-performance computing (HPC) applications

Clustering for massive parallelism

- Of the Top 500 supercomputers reported in 2010, 85 percent were computer clusters or MPPs built with homogeneous nodes.
- Computer clusters have laid the foundation for today's supercomputers, computational grids, and Internet clouds built over data centers.
- We have come a long way toward becoming addicted to computers.
- According to a recent IDC prediction, the HPC market will increase from \$8.5 billion in 2010 to \$10.5 billion by 2013.
- A majority of the Top 500 supercomputers are used for HPC applications in science and engineering. Meanwhile, the use of high throughput computing (HTC) clusters of servers is growing rapidly in business and web services applications.

Cluster Development Trends

- Support for clustering of computers has moved from interconnecting high-end mainframe computers to building clusters with massive numbers of x86 engines. Computer clustering started with the linking of large mainframe computers such as the IBM Sysplex and the SGI Origin 3000.
- Originally, this was motivated by a demand for cooperative group computing and to provide higher availability in critical enterprise applications.
- Subsequently, the clustering trend moved toward the networking of many minicomputers, such as DEC's VMS cluster, in which multiple VAXes were interconnected to share the same set of disk/tape controllers.
- Tandem's Himalaya was designed as a business cluster for fault-tolerant online transaction processing (OLTP) applications.

Cluster Development Trends

- Clustered products now appear as integrated systems, software tools, availability infrastructure, and operating system extensions.
- This clustering trend matches the downsizing trend in the computer industry.
- Supporting clusters of smaller nodes will increase sales by allowing modular incremental growth in cluster configurations.
- From IBM, DEC, Sun, and SGI to Compaq and Dell, the computer industry has leveraged clustering of low-cost servers or x86 desktops for their cost-effectiveness, scalability, and HA features

Milestone Cluster Systems

- A Unix cluster of SMP servers running VMS/OS with extensions, mainly used in high availability applications.
- An AIX server cluster built with Power2 nodes and Omega network and supported by IBM Load leveler and MPI extensions.
- A scalable and fault-tolerant cluster for OLTP and database processing built with non-stop operating system support.
- The Google search engine was built at Google using commodity components. MOSIX is a distributed operating systems for use in Linux clusters, multi-clusters, grids, and the clouds, originally developed by Hebrew University in 1999.

Design Objectives of Computer Clusters

Scalability

- Clustering of computers is based on the concept of modular growth.
- To scale a cluster from hundreds of uniprocessor nodes to a supercluster with 10,000 multicore nodes is a nontrivial task.
- The scalability could be limited by a number of factors, such as the multicore chip technology, cluster topology, packaging method, power consumption, and cooling scheme applied.
- The purpose is to achieve scalable performance constrained by the aforementioned factors.
- We have to also consider other limiting factors such as the memory wall, disk I/O bottlenecks, and latency tolerance, among others.

Design Objectives of Computer Clusters

Packaging

- Cluster nodes can be packaged in a compact or a slack fashion.
- In a compact cluster, the nodes are closely packaged in one or more racks sitting in a room, and the nodes are not attached to peripherals (monitors, keyboards, mice, etc.).
- In a slack cluster, the nodes are attached to their usual peripherals (i.e., they are complete SMPs, workstations, and PCs), and they may be located in different rooms, different buildings, or even remote regions.

Design Objectives of Computer Clusters

- Packaging directly affects communication wire length, and thus the selection of interconnection technology used.
- While a compact cluster can utilize a high-bandwidth, low-latency communication network that is often proprietary, nodes of a slack cluster are normally connected through standard LANs or WANs.

Design Objectives of Computer Clusters

Control

- A cluster can be either controlled or managed in a centralized or decentralized fashion.
- A compact cluster normally has centralized control, while a slack cluster can be controlled either way.
- In a centralized cluster, all the nodes are owned, controlled, managed, and administered by a central operator.
- In a decentralized cluster, the nodes have individual owners.

Design Objectives of Computer Clusters

- For instance, consider a cluster comprising an interconnected set of desktop workstations in a department, where each workstation is individually owned by an employee.
- The owner can reconfigure, upgrade, or even shut down the workstation at any time.
- This lack of a single point of control makes system administration of such a cluster very difficult.
- It also calls for special techniques for process scheduling, workload migration, checkpointing, accounting, and other similar tasks.

Design Objectives of Computer Clusters

Homogeneity

- A homogeneous cluster uses nodes from the same platform, that is, the same processor architecture and the same operating system; often, the nodes are from the same vendors.
- A heterogeneous cluster uses nodes of different platforms.
- Interoperability is an important issue in heterogeneous clusters.
- For instance, process migration is often needed for load balancing or availability.
- In a homogeneous cluster, a binary process image can migrate to another node and continue execution.
- This is not feasible in a heterogeneous cluster, as the binary code will not be executable when the process migrates to a node of a different platform

Design Objectives of Computer Clusters

Security

- Intracluster communication can be either exposed or enclosed.
- In an exposed cluster, the communication paths among the nodes are exposed to the outside world.
- An outside machine can access the communication paths, and thus individual nodes, using standard protocols (e.g., TCP/IP).

Design Objectives of Computer Clusters

- Such exposed clusters are easy to implement, but have several disadvantages:
 - Being exposed, intracluster communication is not secure, unless the communication subsystem performs additional work to ensure privacy and security.
 - Outside communications may disrupt intracluster communications in an unpredictable fashion. For instance, heavy BBS traffic may disrupt production jobs.
 - Standard communication protocols tend to have high overhead.

Design Objectives of Computer Clusters

- In an enclosed cluster, intracluster communication is shielded from the outside world, which alleviates the aforementioned problems.
- A disadvantage is that there is currently no standard for efficient, enclosed intracluster communication.
- Consequently, most commercial or academic clusters realize fast communications through one-of-a-kind protocols.

Design Objectives of Computer Clusters

Dedicated versus Enterprise Clusters

- A dedicated cluster is typically installed in a deskside rack in a central computer room.
- It is homogeneously configured with the same type of computer nodes and managed by a single administrator group like a frontend host.
- Dedicated clusters are used as substitutes for traditional mainframes or supercomputers.
- A dedicated cluster is installed, used, and administered as a single machine.
- Many users can log in to the cluster to execute both interactive and batch jobs.
- The cluster offers much enhanced throughput, as well as reduced response time.

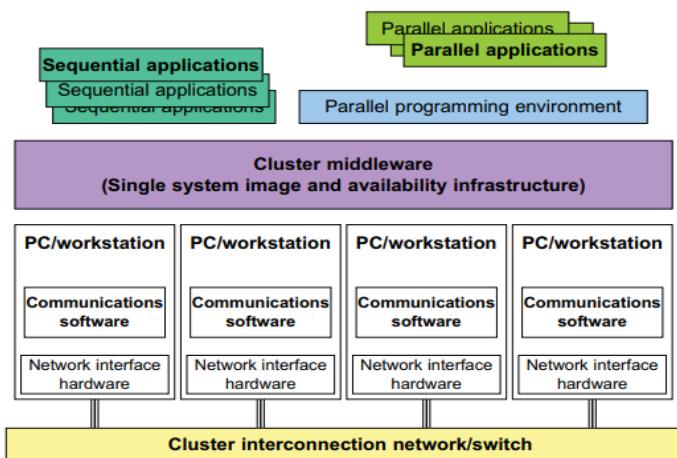
Design Objectives of Computer Clusters

- An enterprise cluster is mainly used to utilize idle resources in the nodes.
- Each node is usually a full-fledged SMP, workstation, or PC, with all the necessary peripherals attached.
- The nodes are typically geographically distributed, and are not necessarily in the same room or even in the same building.
- The nodes are individually owned by multiple owners.
- The cluster administrator has only limited control over the nodes, as a node can be turned off at any time by its owner.
- The owner's "local" jobs have higher priority than enterprise jobs. The cluster is often configured with heterogeneous computer nodes.
- The nodes are often connected through a low-cost Ethernet network. Most data centers are structured with clusters of low-cost servers.

Cluster Organization and Resource Sharing

- The figure shows a simple cluster of computers built with commodity components and fully supported with desired SSI features and HA capability.
- The processing nodes are commodity workstations, PCs, or servers.
- These commodity nodes are easy to replace or upgrade with new generations of hardware.
- The node operating systems should be designed for multiuser, multitasking, and multithreaded applications.
- The nodes are interconnected by one or more fast commodity networks.
- These networks use standard communication protocols and operate at a speed that should be two orders of magnitude faster than that of the current TCP/IP speed over Ethernet.

Cluster Organization and Resource Sharing



The architecture of a computer cluster built with commodity hardware, software, middleware, and network components supporting HA and SSI.

Cluster Organization and Resource Sharing

- When the processor or the operating system is changed, only the driver software needs to change.
- We desire to have a platform-independent cluster operating system, sitting on top of the node platforms.
- But such a cluster OS is not commercially available. Instead, we can deploy some cluster middleware to glue together all node platforms at the user space.
- An availability middleware offers HA services.
- An SSI layer provides a single entry point, a single file hierarchy, a single point of control, and a single job management system. Single memory may be realized with the help of the compiler or a runtime library.
- A single process space is not necessarily supported

Cluster Organization and Resource Sharing

- In general, an idealized cluster is supported by three subsystems.
- First, conventional databases and OLTP monitors offer users a desktop environment in which to use the cluster. In addition to running sequential user programs, the cluster supports parallel programming based on standard languages and communication libraries using PVM, MPI, or OpenMP.
- The programming environment also includes tools for debugging, profiling, monitoring, and so forth.
- A user interface subsystem is needed to combine the advantages of the web interface and the Windows GUI.
- It should also provide user-friendly links to various programming environments, job management tools, hypertext, and search support so that users can easily get help in programming the computer cluster.

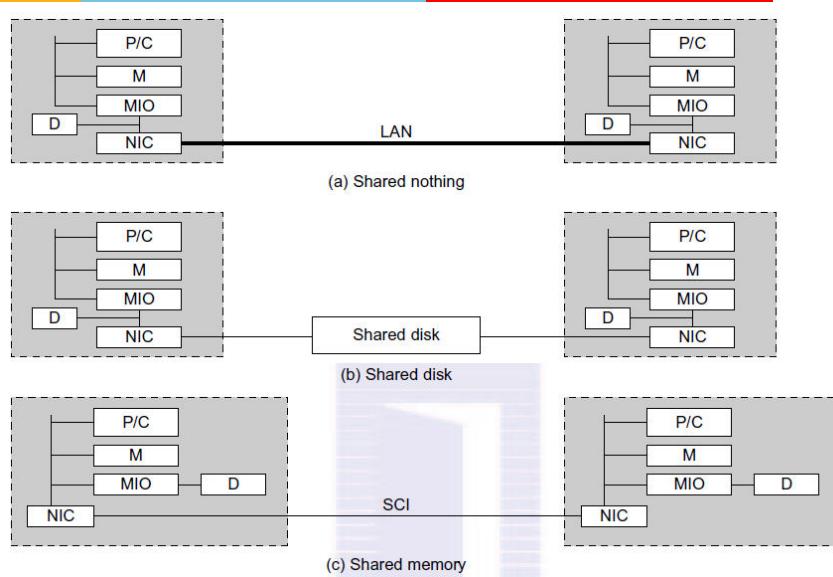
Resource Sharing in Clusters

- Clustering improves both availability and performance. These two clustering goals are not necessarily in conflict.
- Some HA clusters use hardware redundancy for scalable performance. The nodes of a cluster can be connected in one of three ways
- The shared-nothing architecture is used in most clusters, where the nodes are connected through the I/O bus.
- The shared-disk architecture is in favor of small-scale availability clusters in business applications.
- When one node fails, the other node takes over.

Resource Sharing in Clusters

- The shared-nothing configuration simply connects two or more autonomous computers via a LAN such as Ethernet.
- A shared-disk cluster is what most business clusters desire so that they can enable recovery support in case of node failure. The shared disk can hold checkpoint files or critical system images to enhance cluster availability. Without shared disks, checkpointing, rollback recovery, failover, and fallback are not possible in a cluster.
- The shared-memory cluster is much more difficult to realize.
- The nodes could be connected by a scalable coherence interface (SCI) ring, which is connected to the memory bus of each node through an NIC module.
- In the other two architectures, the interconnect is attached to the I/O bus.
- The memory bus operates at a higher frequency than the I/O bus.

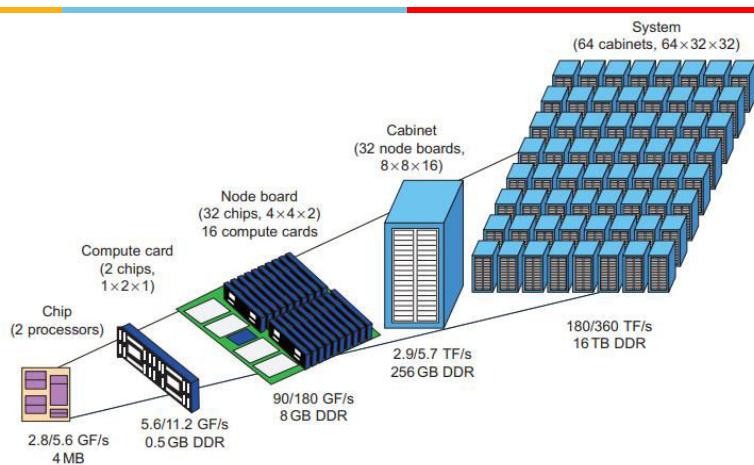
Resource Sharing in Clusters



Node Architectures and MPP Packaging

- In building large-scale clusters or MPP systems, cluster nodes are classified into two categories: compute nodes and service nodes.
- Compute nodes appear in larger quantities mainly used for large-scale searching or parallel floating-point computations.
- Service nodes could be built with different processors mainly used to handle I/O, file access, and system monitoring.
- For MPP clusters, the compute nodes dominate in system cost, because we may have 1,000 times more compute nodes than service nodes in a single large clustered system

Node Architectures and MPP Packaging



The IBM Blue Gene/L architecture built with modular components packaged hierarchically in five levels.

Node Architectures and MPP Packaging

- In the lower-left corner, we see a dual-processor chip. Two chips are mounted on a computer card.
- Sixteen computer cards (32 chips or 64 processors) are mounted on a node board.
- A cabinet houses 32 node boards with an 8 x 8 x 16 torus interconnect.
- Finally, 64 cabinets (racks) form the total system at the upper-right corner.
- This packaging diagram corresponds to the 2005 configuration.
- Customers can order any size to meet their computational needs.
- The Blue Gene cluster was designed to achieve scalable performance, reliability through built-in testability, resilience by preserving locality of failures and checking mechanisms, and serviceability through partitioning and isolation of fault locations.

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self

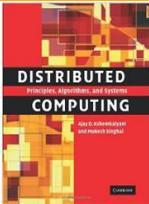


START RECORDING

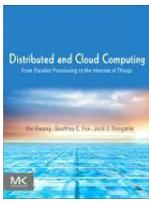


CS-14 : Cluster Computing
[R2: Chap – 2]

Text and References



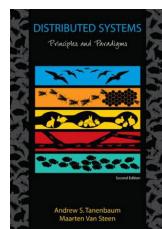
T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).



R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.



R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall

Objective of Module – 9

Cluster Computing

This module introduces Cluster computing as

- (i) a model of carrying out distributed computation using a collection of homogeneous computers and
- (ii) as a building block for Cloud infrastructure.

Contact Session – 14

Cluster Computing

- Cluster system interconnects
- Hardware, software and Middleware support
- GPU Clusters for massive parallelism
- Cluster job and resource management

Presentation Overview

- Cluster System Interconnects
 - Crossbar Switch in Google Search Engine Cluster
 - Share of System Interconnects over Time
- Hardware, Software, and Middleware Support
- GPU Clusters for Massive Parallelism
- Cluster Job Scheduling Methods
 - Space Sharing
 - Time Sharing
 - Independent scheduling
 - Gang scheduling
 - Competition with foreign (local) jobs

Cluster System Interconnects

High-Bandwidth Interconnects

- Ethernet used a 1 Gbps link, while the fastest InfiniBand links ran at 30 Gbps.
- The Myrinet and Quadrics perform in between.
- The MPI latency represents the state of the art in long-distance message passing.
- All four technologies can implement any network topology, including crossbar switches, fat trees, and torus networks.
- The InfiniBand is the most expensive choice with the fastest link speed.
- The Ethernet is still the most cost-effective choice.

Cluster System Interconnects

Feature	Myrinet	Quadrics	InfiniBand	Ethernet
Available link speeds	1.28 Gbps (M-XP) 10 Gbps (M-10G)	2.8 Gbps (QsNet) 7.2 Gbps (QsNetII)	2.5 Gbps (1X) 10 Gbps (4X) 30 Gbps (12X)	1 Gbps
MPI latency	~3 us	~3 us	~4.5 us	~40 us
Network processor	Yes	Yes	Yes	No
Topologies	Any	Any	Any	Any
Network topology	Clos	Fat tree	Fat tree	Any
Routing	Source-based, cut-through	Source-based, cut-through	Destination-based	Destination-based
Flow control	Stop and go	Worm-hole	Credit-based	802.3x

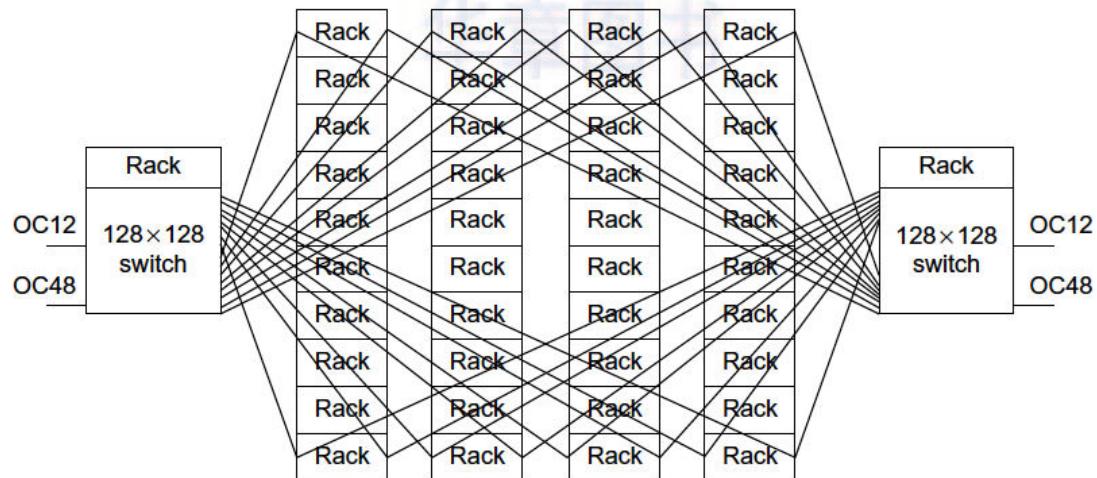
Crossbar Switch in Google Search Engine Cluster

- Google has many data centers using clusters of low-cost PC engines.
- These clusters are mainly used to support Google's web search business.
- Google cluster interconnect of 40 racks of PC engines via two racks of 128 x 128 Ethernet switches.
- Each Ethernet switch can handle 128 one Gbps Ethernet links.
- A rack contains 80 PCs. This is an earlier cluster of 3,200 PCs. Google's search engine clusters are built with a lot more nodes.
- Today's server clusters from Google are installed in data centers with container trucks.

Crossbar Switch in Google Search Engine Cluster

- Two switches are used to enhance cluster availability.
- The cluster works fine even when one switch fails to provide the links among the PCs.
- The front ends of the switches are connected to the Internet via 2.4 Gbps OC 12 links. The 622 Mbps OC 12 links are connected to nearby data-center networks.
- In case of failure of the OC 48 links, the cluster is still connected to the outside world via the OC 12 links.
- Thus, the Google cluster eliminates all single points of failure.

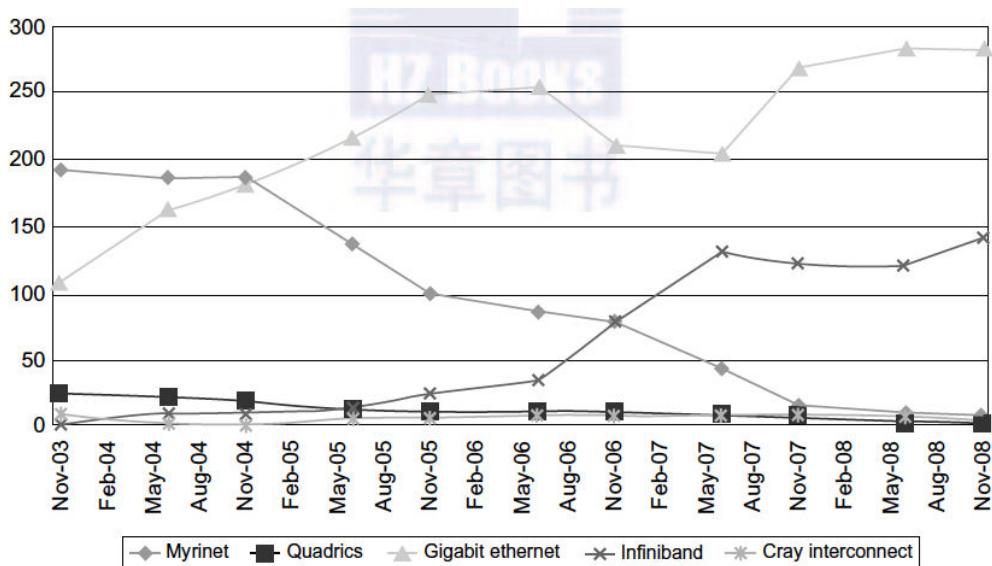
Crossbar Switch in Google Search Engine Cluster



Share of System Interconnects over Time

- The distribution of large-scale system interconnects in the Top 500 systems from 2003 to 2008.
- Gigabit Ethernet is the most popular interconnect due to its low cost and market readiness.
- The InfiniBand network has been chosen in about 150 systems for its high-bandwidth performance.
- The Cray interconnect is designed for use in Cray systems only.
- The use of Myrinet and Quadrics networks had declined rapidly in the Top 500 list by 2008.

Share of System Interconnects over Time



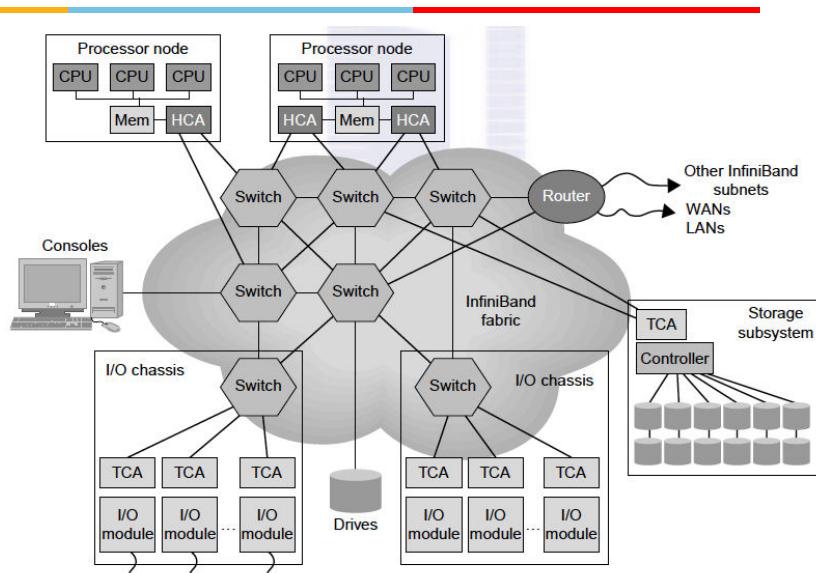
Share of System Interconnects over Time

- The InfiniBand has a switch-based point-to-point interconnect architecture.
- A large InfiniBand has a layered architecture.
- The interconnect supports the virtual interface architecture (VIA) for distributed messaging.
- The InfiniBand switches and links can make up any topology. Popular ones include crossbars, fat trees, and torus networks.
- The InfiniBand provides the highest speed links and the highest bandwidth in reported largescale systems.
- However, InfiniBand networks cost the most among the four interconnect technologies.

Share of System Interconnects over Time

- Each end point can be a storage controller, a network interface card (NIC), or an interface to a host system.
- A host channel adapter (HCA) connected to the host processor through a standard peripheral component interconnect (PCI), PCI extended (PCI-X), or PCI express bus provides the host interface.
- Each HCA has more than one InfiniBand port. A target channel adapter (TCA) enables I/O devices to be loaded within the network.
- The TCA includes an I/O controller that is specific to its particular device's protocol such as SCSI, Fibre Channel, or Ethernet.
- This architecture can be easily implemented to build very large scale cluster interconnects that connect thousands or more hosts together

Share of System Interconnects over Time



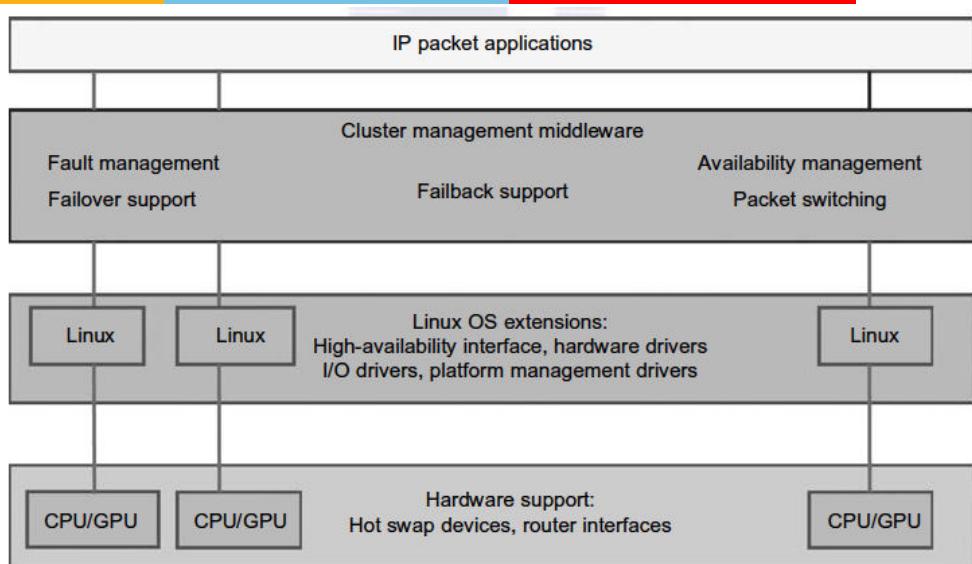
Hardware, Software, and Middleware Support

- Realistically, SSI and HA features in a cluster are not obtained free of charge.
- They must be supported by hardware, software, middleware, or OS extensions.
- Any change in hardware design and OS extensions must be done by the manufacturer.
- The hardware and OS support could be cost prohibitive to ordinary users.
- However, programming level is a big burden to cluster users.
- Therefore, the middleware support at the application level costs the least to implement.

Hardware, Software, and Middleware Support

- Close to the user application end, middleware packages are needed at the cluster management level: one for fault management to support failover and failback
- Another desired feature is to achieve HA using failure detection and recovery and packet switching.
- In the middle of Figure we need to modify the Linux OS to support HA, and we need special drivers to support HA, I/O, and hardware devices.
- Toward the bottom, we need special hardware to support hot-swapped devices and provide router interfaces

Hardware, Software, and Middleware Support



GPU Clusters for Massive Parallelism

- Commodity GPUs are becoming high-performance accelerators for data-parallel computing. Modern GPU chips contain hundreds of processor cores per chip.
- Based on a 2010 report each GPU chip is capable of achieving up to 1 Tflops for single-precision (SP) arithmetic, and more than 80 Gflops for double-precision (DP) calculations.
- Recent HPC-optimized GPUs contain up to 4 GB of on-board memory, and are capable of sustaining memory bandwidths exceeding 100 GB/second.
- GPU clusters are built with a large number of GPU chips.
- GPU clusters have already demonstrated their capability to achieve Pflops performance in some of the Top 500 systems.

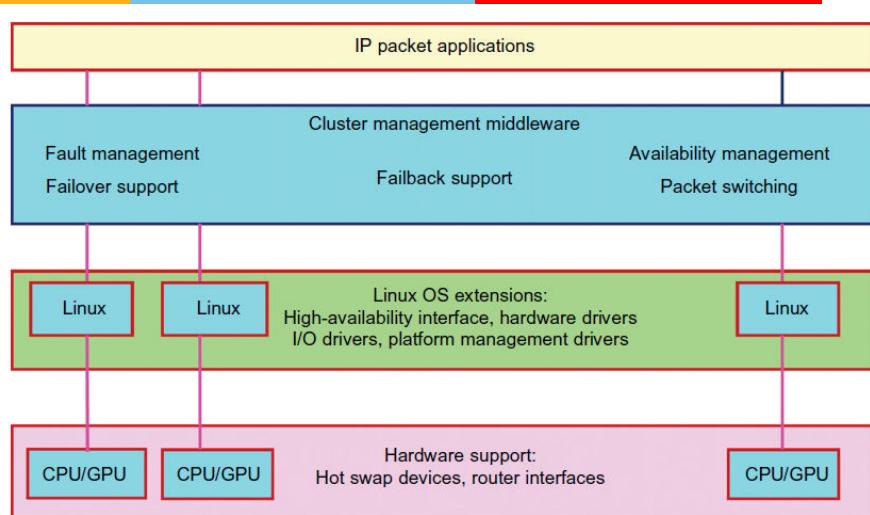
GPU Clusters for Massive Parallelism

- Most GPU clusters are structured with homogeneous GPUs of the same hardware class, make, and model.
- The software used in a GPU cluster includes the OS, GPU drivers, and clustering API such as an MPI.
- The high performance of a GPU cluster is attributed mainly to its massively parallel multicore architecture, high throughput in multithreaded floating-point arithmetic, and significantly reduced time in massive data movement using large on-chip cache memory.
- In other words, GPU clusters already are more cost-effective than traditional CPU clusters.

GPU Clusters for Massive Parallelism

- GPU clusters result in not only a quantum jump in speed performance, but also significantly reduced space, power, and cooling demands.
- A GPU cluster can operate with a reduced number of operating system images, compared with CPU-based clusters.
- These reductions in power, environment, and management complexity make GPU clusters very attractive for use in future HPC applications

GPU Clusters for Massive Parallelism



Middleware, Linux extensions, and hardware support for achieving massive parallelism and HA in a Linux cluster system built with CPUs and GPUs.

Cluster Job Scheduling Methods

- Cluster jobs may be scheduled to run at a specific time (calendar scheduling) or when a particular event happens (event scheduling).
- Jobs are scheduled according to priorities based on submission time, resource nodes, execution time, memory, disk, job type, and user identity.
- With static priority , jobs are assigned priorities according to a predetermined, fixed scheme.
- A simple scheme is to schedule jobs in a first-come, first-serve fashion.
- Another scheme is to assign different priorities to users.
- With dynamic priority , the priority of a job may change over time.

Cluster Job Scheduling Methods

- Three schemes are used to share cluster nodes.
- In the dedicated mode , only one job runs in the cluster at a time, and at most, one process of the job is assigned to a node at a time.
- The single job runs until completion before it releases the cluster to run other jobs.
- Note that even in the dedicated mode, some nodes may be reserved for system use and not be open to the user job.
- Other than that, all cluster resources are devoted to run a single job.
- This may lead to poor system utilization.

Cluster Job Scheduling Methods

- The job resource requirement can be static or dynamic .
- Static scheme fixes the number of nodes for a single job for its entire period.
- Static scheme may underutilize the cluster resource.
- It cannot handle the situation when the needed nodes become unavailable, such as when the workstation owner shuts down the machine
- Dynamic resource allows a job to acquire or release nodes during execution.
- However, it is much more difficult to implement, requiring cooperation between a running job

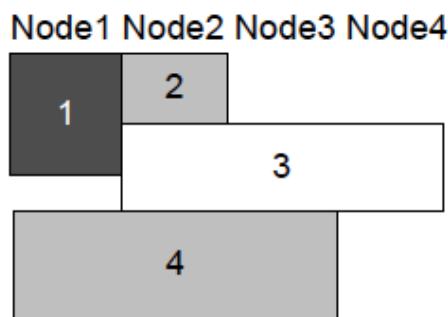
Space Sharing

- A common scheme is to assign higher priorities to short, interactive jobs in daytime and during evening hours using tiling .
- In this space-sharing mode, multiple jobs can run on disjointed partitions (groups) of nodes simultaneously.
- At most, one process is assigned to a node at a time.
- Although a partition of nodes is dedicated to a job, the interconnect and the I/O subsystem may be shared by all jobs.
- Space sharing must solve the tiling problem and the large-job problem.

Space Sharing

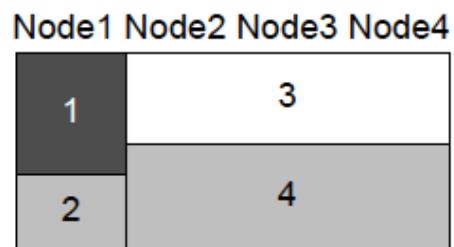
- JMS schedules four jobs in a first-come first-serve fashion on four nodes.
- Jobs 1 and 2 are small and thus assigned to nodes 1 and 2.
- Jobs 3 and 4 are parallel; each needs three nodes.
- When job 3 comes, it cannot run immediately.
- It must wait until job 2 finishes to free up the needed nodes.
- Tiling will increase the utilization of the nodes
- The overall execution time of the four jobs is reduced after repacking the jobs over the available nodes

Space Sharing



(a) First-come first-serve

Time



(b) After tiling

Time Sharing

- In the dedicated or space-sharing model, only one user process is allocated to a node.
- However, the system processes or daemons are still running on the same node.
- In the time-sharing mode, multiple user processes are assigned to the same node.
- Time sharing introduces the following parallel scheduling policies:
 - Independent scheduling
 - Gang scheduling
 - Competition with foreign (local) jobs

Independent scheduling

- The most straightforward implementation of time sharing is to use the operating system of each cluster node to schedule different processes as in a traditional workstation.
- This is called local scheduling or independent scheduling.
- However, the performance of parallel jobs could be significantly degraded.
- Processes of a parallel job need to interact.
- For instance, when one process wants to barrier-synchronize with another, the latter may be scheduled out. So the first process has to wait.
- As the second process is rescheduled, the first process may be swapped out.

Gang scheduling

- The gang scheduling scheme schedules all processes of a parallel job together.
- When one process is active, all processes are active.
- The cluster nodes are not perfectly clock-synchronized.
- In fact, most clusters are asynchronous systems, and are not driven by the same clock.
- Although we say, “All processes are scheduled to run at the same time,” they do not start exactly at the same time

Competition with foreign (local) jobs

- Scheduling becomes more complicated when both cluster jobs and local jobs are running.
- Local jobs should have priority over cluster jobs.
- With one keystroke, the owner wants command of all workstation resources.
- There are basically two ways to deal with this situation:
 - The cluster job can either stay in the workstation node or
 - migrate to another idle node.

Competition with foreign (local) jobs

- A stay scheme has the advantage of avoiding migration cost.
- The cluster process can be run at the lowest priority.
- The workstation's cycles can be divided into three portions, for kernel processes, local processes, and cluster processes.
- However, to stay slows down both the local and the cluster jobs, especially when the cluster job is a load balanced parallel job that needs frequent synchronization and communication.
- This leads to the migration approach to flow the jobs around available nodes, mainly for balancing the workload.

Cluster Job Management Systems

- Job management is also known as workload management, load sharing, or load management.
- A Job Management System (JMS) should have three parts:
 - ✓ A **user server** lets the user submit jobs to one or more queues, specify resource requirements for each job, delete a job from a queue, and inquire about the status of a job or a queue.
 - ✓ A **job scheduler** performs job scheduling and queuing according to job types, resource requirements, resource availability, and scheduling policies.
 - ✓ A **resource manager** allocates and monitors resources, enforces scheduling policies, and collects accounting information.

Cluster Job Management Systems

JMS Administration

- The functionality of a JMS is often distributed. For instance, a user server may reside in each host node, and the resource manager may span all cluster nodes.
- However, the administration of a JMS should be centralized.
- All configuration and log files should be maintained in one location.
- There should be a single user interface to use the JMS.
- It is undesirable to force the user to run PVM jobs through one software package, MPI jobs through another, and HPF jobs through yet another.
- The JMS should be able to dynamically reconfigure the cluster with minimal impact on the running jobs.

Cluster Job Management Systems

- The administrator's prologue and epilogue scripts should be able to run before and after each job for security checking, accounting, and cleanup.
- Users should be able to cleanly kill their own jobs.
- The administrator or the JMS should be able to cleanly suspend or kill any job.
- Clean means that when a job is suspended or killed, all its processes must be included.
- Otherwise, some "orphan" processes are left in the system, which wastes cluster resources and may eventually render the system unusable.

Cluster Job Management Systems

Cluster Job Types

- Several types of jobs execute on a cluster.
- **Serial jobs** run on a single node.
- **Parallel jobs** use multiple nodes.
- **Interactive jobs** are those that require fast turnaround time, and their input/output is directed to a terminal.
- These jobs do not need large resources, and users expect them to execute immediately, not to wait in a queue.

Cluster Job Management Systems

- **Batch jobs** normally need more resources, such as large memory space and long CPU time.
- But they do not need immediate responses.
- They are submitted to a job queue to be scheduled to run when the resource becomes available (e.g., during off hours).
- While both interactive and batch jobs are managed by the JMS, **foreign jobs** are created outside the JMS.
- For instance, when a network of workstations is used as a cluster, users can submit interactive or batch jobs to the JMS.

Cluster Job Management Systems

- Meanwhile, the owner of a workstation can start a foreign job at any time, which is not submitted through the JMS.
- Such a job is also called a **local job**, as opposed to cluster jobs (interactive or batch, parallel or serial) that are submitted through the JMS of the cluster.
- The characteristic of a local job is fast response time.
- The owner wants all resources to execute his job, as though the cluster jobs did not exist.

Cluster Job Management Systems

Characteristics of a Cluster Workload

- Roughly half of parallel jobs are submitted during regular working hours. Almost 80 percent of parallel jobs run for three minutes or less. Parallel jobs running longer than 90 minutes account for 50 percent of the total time.
- The sequential workload shows that 60 percent to 70 percent of workstations are available to execute parallel jobs at any time, even during peak daytime hours.
- On a workstation, 53 percent of all idle periods are three minutes or less, but 95 percent of idle time is spent in periods of time that are 10 minutes or longer.
- A 2:1 rule applies, which says that a network of 64 workstations, with proper JMS software, can sustain a 32-node parallel workload in addition to the original sequential workload. In other words, clustering gives a supercomputer half of the cluster size for free!

Cluster Job Management Systems

Migration Schemes

A migration scheme must consider the following three issues:

Node availability

- This refers to node availability for job migration.
- The Berkeley NOW project has reported such opportunity does exist in university campus environment.
- Even during peak hours, 60 percent of workstations in a cluster are available at Berkeley campus.

Cluster Job Management Systems

Migration overhead

- What is the effect of the migration overhead? The migration time can significantly slow down a parallel job.
- It is important to reduce the migration overhead (e.g., by improving the communication subsystem) or to migrate only rarely.
- The slowdown is significantly reduced if a parallel job is run on a cluster of twice the size.
- For instance, for a 32-node parallel job run on a 60-node cluster, the slowdown caused by migration is no more than 20 percent, even when the migration time is as long as three minutes.
- This is because more nodes are available, and thus the migration demand is less frequent.

Cluster Job Management Systems

Recruitment threshold

- What should be the recruitment threshold? In the worst scenario, right after a process migrates to a node, the node is immediately claimed by its owner.
- Thus, the process has to migrate again, and the cycle continues.
- The recruitment threshold is the amount of time a workstation stays unused before the cluster considers it an idle node.

THANK YOU

IMP Note to Self



STOP RECORDING





BITS Pilani

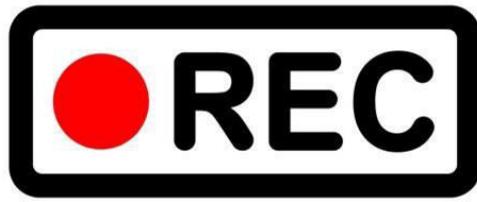
Pilani | Dubai | Goa | Hyderabad

CC ZG526

Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



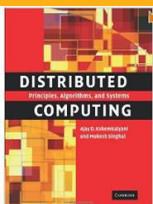
START RECORDING

CC ZG526 Distributed Computing || Week - 15 Dt: 11th May 2025 2 BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

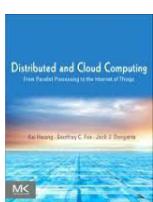
CS-15 : Distributed data

[R3: Chap – 7]

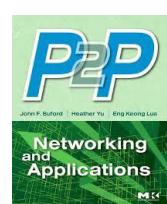
Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

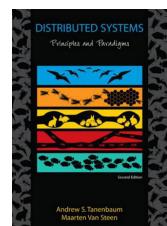


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 10

Distributed data

- Distributed systems need to provide a view of data consistency at a global level or specific to clients connecting to the system.
- This module will discuss replication, various types of consistency and protocols.
- Practical examples will be discussed through case studies of modern distributed databases.

Contact Session – 15

Consistency in Distributed Systems

- CAP theorem and implications in modern distributed systems
- Data centric consistency
- Client centric consistency
- Replica management
- Consistency protocols

Presentation Overview

- CAP theorem
- Reasons for Replication
- Data-centric Consistency Models
 - Consistent Ordering of Operations
- Client-Centric Consistency Models

CAP Theorem

- When designing a distributed system, you might assume that the logical approach is to design it for all three of the most important factors:
 - Consistency
 - Availability
 - Partition tolerance over the network
- For a long time, so did architects and developers.
- Although the CAP theorem approach seemed sensible, a better way was later discovered.

CAP Theorem

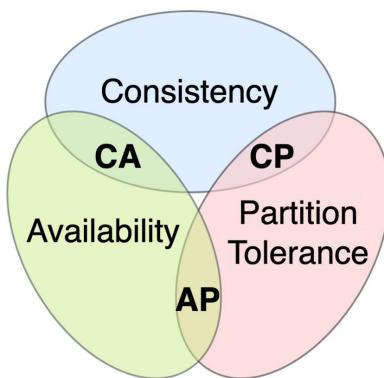
- The CAP theorem is about how it was impossible for a distributed system to simultaneously maintain Consistency, Availability, and Partition Tolerance, hence the name CAP.
- The theorem was introduced by computer scientist Eric Brewer which is why it is sometimes called 'Brewer's theorem'.
- He emphasized that if the system were to be optimized, it should focus on two of the mentioned factors — and compromise the third one.

CAP Theorem

CAP theorem trade-offs

- The real gist of the CAP Theorem is that it is challenging for a distributed system to simultaneously achieve all three characteristics — consistency, availability, and partition tolerance — at their maximum level.
- Therefore, we have to prioritize two characteristics compromising on the third one.
- This leads to three primary trade-offs.
- Be wary of oversimplifying these three compromises, however.
- In practice, the trade-offs are more nuanced.
- System designers frequently make dynamic adjustments based on the specific needs and circumstances of their systems.

CAP Theorem



CAP Theorem

CP systems: Consistency & partition tolerance

- In a system where Consistency and Partition tolerance are prioritized, the system makes sure that the nodes in the network have a constant view of the data, even during network disruptions (aka network partitions).
- This means that when a value is written to one node, the value will be updated in all the other nodes.
- However, CP systems may compromise on availability, meaning that some data might not be accessible in the event of a network partition.

CAP Theorem

CA systems: Consistency & availability

- CA systems prioritize a constant view of the data and ensure the availability is high under normal operating conditions.
- This means that every request made to a (non-faulty) node in the system must return either a successful or a failed response.
- However, CA systems may struggle with network partitions, potentially becoming less available or exhibiting lower performance during such events.
- In reality, pure CA systems are rare.
- That's because pretty much every distributed systems must handle network partitions to some degree.

CAP Theorem

AP systems: Availability & partition tolerance

- AP systems prioritize availability and partition tolerance, rather than strong consistency.
- This means that the system always aims to process queries and provide the most recent available version of information, even if it cannot ensure it is completely up-to-date due to network partitioning.
- In such systems, temporary discrepancies in data versions among different nodes are acceptable, with the primary goal being to ensure high availability.

CAP Theorem

Real-world applications & examples

- Understanding how the CAP theorem applies in real-world scenarios is crucial for making informed decisions when designing distributed systems.

Google Bigtable

- Google Bigtable is a distributed storage system used for managing structured data. It is designed to prioritize consistency and partition tolerance (CP), which means that in the event of a network partition or failure, Bigtable may compromise availability to maintain data consistency.
- However, Google has engineered its network to minimize the occurrence of network partitions, which allows Bigtable to achieve high availability in practice, despite its CP categorization.

CAP Theorem

Amazon DynamoDB

- DynamoDB is a database service provided by AWS designed to prioritize high availability and partition tolerance, in line with the CAP theorem. Initially, it offered only 'eventual consistency,' but now it also provides a 'strong consistency' option.
- This means, that when you ask DynamoDB for the latest data through a consistent read request, it returns a response that is very much up-to-date. It's designed to handle network partitions effectively, maintaining high availability and providing consistent reads as needed.
- Users have the flexibility to choose between eventual and strong consistency, allowing DynamoDB to cater to different requirements while adhering to the CAP theorem principles.

CAP Theorem

MongoDB

- MongoDB is a well-known Database Management System that is frequently used to run real-time applications from different locations.
- This DBMS follows a CP trade-off as it prioritizes Consistency and Partition tolerance over Availability.

CAP Theorem

Apache Cassandra

- Cassandra is a free and open-source, distributed, wide-column store, NoSQL database management system.
- It is designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.
- Cassandra allows clients to specify the number of servers that a write must go to before it is committed on a per-write basis.
- Writes going to a single server are "AP", writes going to quorum (or all) servers are more "CA".

Reasons for Replication

- An important issue in distributed systems is the replication of data.
- Data are generally replicated to enhance reliability or improve performance.
- One of the major problems is keeping replicas consistent.
- There are two primary reasons for replicating data:
 - reliability
 - performance

Reasons for Replication

- First, data are replicated to increase the reliability of a system.
- If a file system has been replicated it may be possible to continue working after one replica crashes by simply switching to one of the other replicas.
- Also, by maintaining multiple copies, it becomes possible to provide better protection against corrupted data.
- For example, imagine there are three copies of a file and every read and write operation is performed on each copy.
- We can safeguard ourselves against a single, failing write operation, by considering the value that is returned by at least two copies as being the correct one.

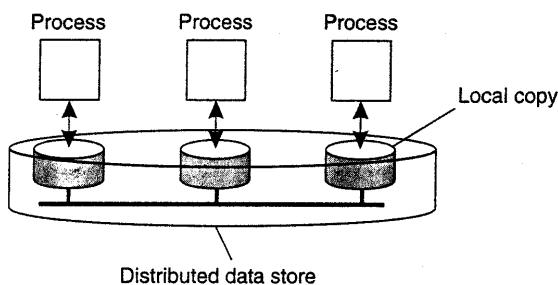
Reasons for Replication

- The other reason for replicating data is performance.
- Replication for performance is important when the distributed system needs to scale in numbers and geographical area.
- Scaling in numbers occurs, for example, when an increasing number of processes needs to access data that are managed by a single server.
- In that case, performance can be improved by replicating the server and subsequently dividing the work.

Data-centric Consistency Models

- A data store may be physically distributed across multiple machines.
- In particular, each process that can access data from the store is assumed to have a local (or nearby) copy available of the entire store.
- Write operations are propagated to the other copies.
- A data operation is classified as a write operation when it changes the data, and is otherwise classified as a read operation.

Data-centric Consistency Models



The general organization of a logical data store, physically distributed and replicated across multiple processes.

Data-centric Consistency Models

- A consistency model is essentially a contract between processes and the data store.
- It says that if processes agree to obey certain rules, the store promises to work correctly.
- Normally, a process that performs a read operation on a data item, expects the operation to return a value that shows the results of the last write operation on that data.
- In the absence of a global clock, it is difficult to define precisely which write operation is the last one.

Data-centric Consistency Models

- As an alternative, we need to provide other definitions, leading to a range of consistency models.
- Each model effectively restricts the values that a read operation on a data item can return.
- As is to be expected, the ones with major restrictions are easy to use, for example when developing applications, whereas those with minor restrictions are sometimes difficult.
- The tradeoff is, of course, that the easy-to-use models do not perform nearly as well as the difficult ones.

Data-centric Consistency Models

Continuous Consistency

- There are different ways for applications to specify what inconsistencies they can tolerate.
- Yu and Vahdat (2002) take a general approach by distinguishing three independent axes for defining inconsistencies:
 - deviation in numerical values between replicas,
 - deviation in staleness between replicas, and
 - deviation with respect to the ordering of update operations.
- They refer to these deviations as forming continuous consistency ranges.

Data-centric Consistency Models

- Measuring inconsistency in terms of numerical deviations can be used by applications for which the data have numerical semantics.
- One obvious example is the replication of records containing stock market prices.
- In this case, an application may specify that two copies should not deviate more than \$0.02, which would be an absolute numerical deviation.
- Alternatively, a relative numerical deviation could be specified, stating that two copies should differ by no more than, for example, 0.5%.
- In both cases, we would see that if a stock goes up (and one of the replicas is immediately updated) without violating the specified numerical deviations, replicas would still be considered to be mutually consistent.

Data-centric Consistency Models

- Numerical deviation can also be understood in terms of the number of updates that have been applied to a given replica, but have not yet been seen by others.
- For example, a Web cache may not have seen a batch of operations carried out by a Web server.
- In this case, the associated deviation in the value is also referred to as its weight.

Data-centric Consistency Models

- Staleness deviations relate to the last time a replica was updated.
- For some applications, it can be tolerated that a replica provides old data as long as it is not too old.
- For example, weather reports typically stay reasonably accurate over some time, say a few hours. In such cases, a main server may receive timely updates, but may decide to propagate updates to the replicas only once in a while.

Data-centric Consistency Models

- Finally, there are classes of applications in which the ordering of updates are allowed to be different at the various replicas, as long as the differences remain bounded.
- One way of looking at these updates is that they are applied tentatively to a local copy, awaiting global agreement from all replicas.
- As a consequence, some updates may need to be rolled back and applied in a different order before becoming permanent.
- Intuitively, ordering deviations are much harder to grasp than the other two consistency metrics.

Consistent Ordering of Operations

Sequential Consistency

- In the following, we will use a special notation in which we draw the operations of a process along a time axis.
- The time axis is always drawn horizontally, with time increasing from left to right.

$W_i(x)a$ and $R_i(x)b$

P1:	W(x)a
P2:	R(x)NIL R(x)a

Consistent Ordering of Operations

- As an example, P1 does a write to a data item x, modifying its value to a.
- Note that, in principle, this operation $W_1(x)a$ is first performed on a copy of the data store that is local to P1, and is then subsequently propagated to the other local copies.
- In our example, P2 later reads the value NIL, and some time after that a (from its local copy of the store).
- What we are seeing here is that it took some time to propagate the update of x to P2, which is perfectly acceptable.
- Sequential consistency is an important data-centric consistency model, which was first defined by Lamport (1979) in the context of shared memory for multiprocessor systems.

Consistent Ordering of Operations

In general, a data store is said to be sequentially consistent when it satisfies the following condition:

The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.

Consistent Ordering of Operations

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b R(x)a
P4:	R(x)b R(x)a

(a)

P1:	W(x)a
P2:	W(x)b
P3:	R(x)b R(x)a
P4:	R(x)a R(x)b

(b)

- (a) A sequentially consistent data store.
(b) (b) A data store that is not sequentially consistent.

Consistent Ordering of Operations

Causal Consistency

- The causal consistency model (Hutto and Ahamad, 1990) represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not.
- If event b is caused or influenced by an earlier event a, causality requires that everyone else first see a, then see b.
- Consider a simple interaction by means of a distributed shared database.
- Suppose that process P, writes a data item x.
- Then P2 reads x and writes y.
- Here the reading of x and the writing of y are potentially causally related because the computation of y may have depended on the value of x as read by P2.

Consistent Ordering of Operations

- On the other hand, if two processes spontaneously and simultaneously write two different data items, these are not causally related.
- Operations that are not causally related are said to be concurrent.
- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

Consistent Ordering of Operations

Grouping Operations

- Sequential and causal consistency are defined at the level read and write operations.
- This level of granularity is for historical reasons, these models have initially been developed for shared-memory multiprocessor systems and were actually implemented at the hardware level.

Consistent Ordering of Operations

- The fine granularity of these consistency models in many cases did not match the granularity as provided by applications.
- What we see there is that concurrency between programs sharing data is generally kept under control through synchronization mechanisms for mutual exclusion and transactions.
- Effectively, what happens is that at the program level read and write operations are bracketed by the pair of operations ENTER_CS and LEAVE_CS where "CS" stands for critical section.

Consistent Ordering of Operations

We now demand that the following criteria are met (Bershad et al., 1993):

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process' next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

Client-Centric Consistency Models

- The data stores we consider are characterized by the lack of simultaneous updates, or when such updates happen, they can easily be resolved.
- Most operations involve reading data.
- These data stores offer a very weak consistency model, called eventual consistency.
- By introducing special client-centric consistency models, it turns out that many inconsistencies can be hidden in a relatively cheap way.

Eventual Consistency

- To what extent processes actually operate in a concurrent fashion, and to what extent consistency needs to be guaranteed, may vary.
- There are many examples in which concurrency appears only in a restricted form.
- For example, in many database systems, most processes hardly ever perform update operations; they mostly read data from the database.
- Only one, or very few processes perform update operations.
- The question then is how fast updates should be made available to only reading processes.

Eventual Consistency

- Data stores that are eventually consistent thus have the property that in the absence of updates, all replicas converge toward identical copies of each other.
- Eventual consistency essentially requires only that updates are guaranteed to propagate to all replicas.
- Write-write conflicts are often relatively easy to solve when assuming that only a small group of processes can perform updates.
- Eventual consistency is therefore often cheap to implement.

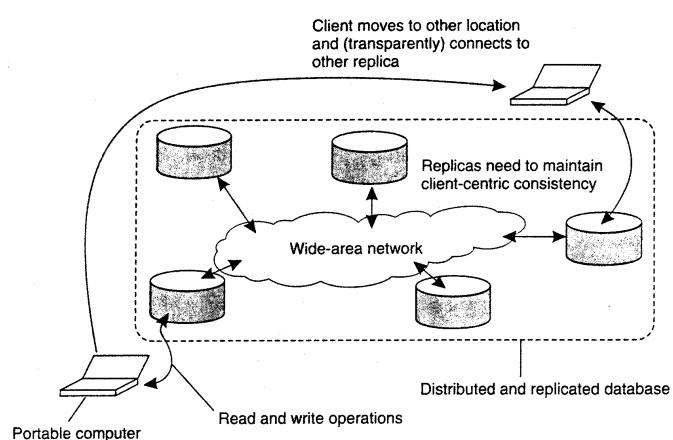
Eventual Consistency

- Eventual consistent data stores work fine as long as clients always access the same replica.
- However, problems arise when different replicas are accessed over a short period of time.
- This is best illustrated by considering a mobile user accessing a distributed database

Eventual Consistency

- The mobile user accesses the database by connecting to one of the replicas in a transparent way.
- In other words, the application running on the user's portable computer is unaware on which replica it is actually operating.
- Assume the user performs several update operations and then disconnects again.
- Later, he accesses the database again, possibly after moving to a different location or by using a different access device.
- At that point, the user may be connected to a different replica than before.
- However, if the updates performed previously have not yet been propagated, the user will notice inconsistent behavior.
- In particular, he would expect to see all previously made changes, but instead, it appears as if nothing at all has happened.

Eventual Consistency



The principle of a mobile user accessing different replicas of a distributed database.

Eventual Consistency

- This example is typical for eventually-consistent data stores and is caused by the fact that users may sometimes operate on different replicas.
- The problem can be alleviated by introducing client-centric consistency.
- In essence, client-centric consistency provides guarantees for a single client concerning the consistency of accesses to a data store by that client.
- No guarantees are given concerning concurrent accesses by different clients.

Monotonic Reads

- The first client-centric consistency model is that of monotonic reads.
- A data is said to provide monotonic-read consistency if the following condition holds:

If a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

- In other words, monotonic-read consistency guarantees that if a process has seen a value of x at time t, it will never see an older version of x at a later time.

Monotonic Writes

- In many situations, it is important that write operations are propagated in the correct order to all copies of the data store.
- This property is expressed in monotonic-write consistency.
- In a monotonic-write consistent store, the following condition holds:

A write operation by a process on a data item x is completed before any successive write operation on X by the same process.

Monotonic Writes

- Thus completing a write operation means that the copy on which a successive operation is performed reflects the effect of a previous write operation by the same process, no matter where that operation was initiated.
- In other words, a write operation on a copy of item x is performed only if that copy has been brought up to date by means of any preceding write operation, which may have taken place on other copies of x.
- If need be, the new write must wait for old ones to finish.

Read Your Writes

- A client-centric consistency model that is closely related to monotonic reads is as follows.
- A data store is said to provide read-your-writes consistency, if the following condition holds:

The effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process.

- In other words, a write operation is always completed before a successive read operation by the same process, no matter where that read operation takes place.

Writes Follow Reads

- The last client-centric consistency model is one in which updates are propagated as the result of previous read operations.
- A data store is said to provide writes-follow-reads consistency, if the following holds.

A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

- In other words, any successive write operation by a process on a data item x will be performed on a copy of x that is up to date with the value most recently read by that process.

THANK YOU

IMP Note to Self



STOP RECORDING



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

CC ZG526
Distributed Computing

Anil Kumar Ghadiyaram
ganilkumar@wilp.bits-pilani.ac.in

IMP Note to Self



START RECORDING

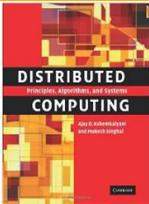
BITS Pilani
Pilani | Dubai | Goa | Hyderabad



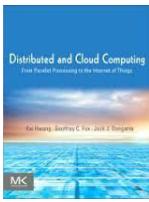
CS-16 : Distributed data

[R3: Chap – 7]

Text and References



T1 - Ajay D. Kshemkalyani, and Mukesh Singhal "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2008 (Reprint 2013).

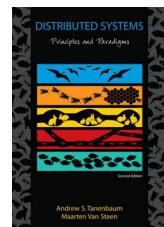


R1 - John F. Buford, Heather Yu, and Eng K. Lua, "P2P Networking and Applications", Morgan Kaufmann, 2009 Elsevier Inc.



R2 - Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, "Distributed and Cloud Computing: From Parallel processing to the Internet of Things", Morgan Kaufmann, 2012 Elsevier Inc.

R3 - A. Tanenbaum and M. V. Steen, "Distributed Systems", 2nd Edition, Pearson Prentice Hall



Objective of Module – 10

Distributed data

- Distributed systems need to provide a view of data consistency at a global level or specific to clients connecting to the system.
- This module will discuss replication, various types of consistency and protocols.
- Practical examples will be discussed through case studies of modern distributed databases.

Presentation Overview

- Replica management
 - Replica-Server Placement
 - Content Replication and Placement
 - Content Distribution
- Consistency protocols
 - Primary-Based Protocols
 - Replicated-Write Protocols
 - Cache-Coherence Protocols

Replica Management

- A key issue for any distributed system that supports replication is to decide where, when, and by whom replicas should be placed, and subsequently which mechanisms to use for keeping the replicas consistent.
- The placement problem itself should be split into two subproblems:
 - that of placing replica servers
 - that of placing content
- The difference is a subtle but important one and the two issues are often not clearly separated.

Replica Management

- Replica-server placement is concerned with finding the best locations to place a server that can host (part of) a data store.
- Content placement deals with finding the best servers for placing content.
- Note that this often means that we are looking for the optimal placement of only a single data item.
- Obviously, before content placement can take place, replica servers will have to be placed first.

Replica-Server Placement

- There are various ways to compute the best placement of replica servers, but all boil down to an optimization problem in which the best K out of N locations need to be selected ($K < N$).
- These problems are known to be computationally complex and can be solved only through heuristics.
- Qiu et al. (2001) take the distance between clients and locations as their starting point.
- Distance can be measured in terms of latency or bandwidth.
- Their solution selects one server at a time such that the average distance between that server and its clients is minimal given that already k servers have been placed (meaning that there are $N - k$ locations left).

Replica-Server Placement

- As an alternative, Radoslavov et al. (2001) propose to ignore the position of clients and only take the topology of the Internet as formed by the autonomous systems.
- An autonomous system (AS) can best be viewed as a network in which the nodes all run the same routing protocol and which is managed by a single organization.
- One problem with these algorithms is that they are computationally expensive.

Replica-Server Placement

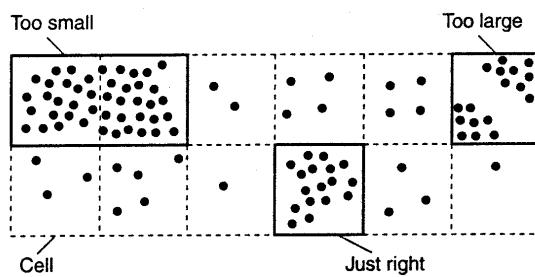
- Szymaniak et al. (2006) have developed a method by which a region for placing replicas can be quickly identified.
- A region is identified to be a collection of nodes accessing the same content, but for which the internode latency is low.
- The goal of the algorithm is first to select the most demanding regions—that is, the one with the most nodes—and then to let one of the nodes in such a region act as replica server.

Replica-Server Placement

- To this end, nodes are assumed to be positioned in an m -dimensional geometric space.
- The basic idea is to identify the K largest clusters and assign a node from each cluster to host replicated content.
- To identify these clusters, the entire space is partitioned into cells.
- The K most dense cells are then chosen for placing a replica server.
- A cell is nothing but an m -dimensional hypercube.
- For a two-dimensional space, this corresponds to a rectangle.

Replica-Server Placement

- Obviously, the cell size is important, if cells are chosen too large, then multiple clusters of nodes may be contained in the same cell.
- In that case, too few replica servers for those clusters would be chosen.
- On the other hand, choosing small cells may lead to the situation that a single cluster is spread across a number of cells, leading to choosing too many replica servers.



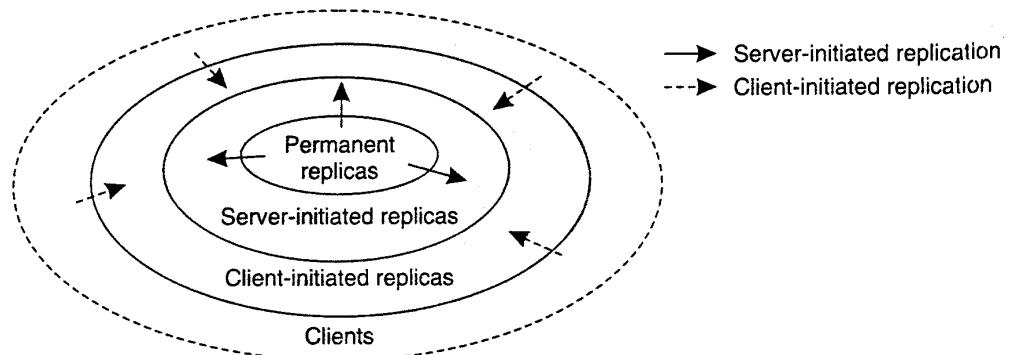
Replica-Server Placement

- As it turns out, an appropriate cell size can be computed as a simple function of the average distance between two nodes and the number of required replicas.
- With this cell size, it can be shown that the algorithm performs as well as the close-to-optimal one, but having a much lower complexity:
- Experiments show that computing the 20 best replica locations for a collection of 64,000 nodes is approximately 50.000 times faster.
- As a consequence, replica server placement can now be done in real time.

Content Replication and Placement

- When it comes to content replication and placement, three different types of replicas can be distinguished logically organized
 - Permanent Replicas
 - Server-Initiated Replicas
 - Client-Initiated Replicas

Content Replication and Placement



The logical organization of different kinds of copies of a data store into three concentric rings.

Content Replication and Placement

Permanent Replicas

- Permanent replicas can be considered as the initial set of replicas that constitute a distributed data store. In many cases, the number of permanent replicas is small.
- Database can be distributed and replicated across number of servers that together form a cluster of servers, often referred to as shared-nothing architecture, emphasizing that neither disks nor main memory are shared by processors.
- Alternatively, a database is distributed and possibly replicated across a number of geographically dispersed sites.
- This architecture is generally deployed in federated databases

Content Replication and Placement

Server-Initiated Replicas

- In contrast to permanent replicas, server-initiated replicas are copies of a data store that exist to enhance performance and which are created at the initiative of the (owner of the) data store.
- Consider, for example, a Web server placed in New York. Normally, this server can handle incoming requests quite easily, but it may happen that over a couple of days a sudden burst of requests come in from an unexpected location far from the server.

Content Replication and Placement

- In that case, it may be worthwhile to install a number of temporary replicas in regions where requests are coming from.
- The problem of dynamically placing replicas is also being addressed in Web hosting services.
- These services offer a (relatively static) collection of servers spread across the Internet that can maintain and provide access to Web files belonging to third parties.
- To provide optimal facilities such hosting services can dynamically replicate files to servers where those files are needed to enhance performance, that is, close to demanding (groups of) clients.

Content Replication and Placement

Client-Initiated Replicas

- An important kind of replica is the one initiated by a client.
- Client-initiated replicas are more commonly known as (client) caches.
- In essence, a cache is a local storage facility that is used by a client to temporarily store a copy of the data it has just requested. In principle, managing the cache is left entirely to the client.
- The data store from where the data had been fetched has nothing to do with keeping cached data consistent.
- However, as we shall see, there are many occasions in which the client can rely on participation from the data store to inform it when cached data has become stale.

Content Replication and Placement

- Client caches are used only to improve access times to data.
- Normally, when a client wants access to some data, it connects to the nearest copy of the data store from where it fetches the data it wants to read, or to where it stores the data it had just modified.
- When most operations involve only reading data, performance can be improved by letting the client store requested data in a nearby cache.
- Such a cache could be located on the client's machine, or on a separate machine in the same local-area network as the client.
- The next time that same data needs to be read, the client can simply fetch it from this local cache.
- This scheme works fine as long as the fetched data have not been modified in the meantime. Data are generally kept in a cache for a limited amount of time.

Content Distribution

- Replica management also deals with propagation of (updated) content to the relevant replica servers.
- There are various trade-offs to make, with the following type
 - State versus Operations
 - Pull versus Push Protocols
 - Unicasting versus Multicasting

Content Distribution

State versus Operations

An important design issue concerns what is actually to be propagated. Basically, there are three possibilities:

1. Propagate only a notification of an update.
2. Transfer data from one copy to another.
3. Propagate the update operation to other copies.

Content Distribution

- Consider, for example, a data store in which updates are propagated by sending the modified data to all replicas.
- If the size of the modified data is large, and updates occur frequently compared to read operations, we may have the situation that two updates occur after one another without any read operation being performed between them.
- Consequently, propagation of the first update to all replicas is effectively useless, as it will be overwritten by the second update. Instead, sending a notification that the data have been modified would have been more efficient.

Content Distribution

Pull versus Push Protocols

- In a push based approach, also referred to as server-based protocols, updates are propagated to other replicas without those replicas even asking for the updates.
- Push-based approaches are often used between permanent and server-initiated replicas, but can also be used to push updates to client caches.
- Server-based protocols are applied when replicas generally need to maintain a relatively high degree of consistency.
- In other words, replicas need to be kept identical.

Content Distribution

- In contrast, in a pull-based approach, a server or client requests another server to send it any updates it has at that moment. Pull-based protocols, also called client-based protocols, are often used by client caches.
- For example, a common strategy applied to Web caches is first to check whether cached data items are still up to date.
- When a cache receives a request for items that are still locally available, the cache checks with the original Web server whether those data items have been modified since they were cached.
- In the case of a modification, the modified data are first transferred to the cache, and then returned to the requesting client.
- If no modifications took place, the cached data are returned. In other words, the client polls the server to see whether an update is needed.

Content Distribution

Unicasting versus Multicasting

- Related to pushing or pulling updates is deciding whether unicasting or multicasting should be used.
- In unicast communication, when a server that is part of the data store sends its update to N other servers, it does so by sending N separate messages, one to each server.
- With multicasting, the underlying network takes care of sending a message efficiently to multiple receivers.

Content Distribution

- Multicasting can often be efficiently combined with a push-based approach to propagating updates.
- When the two are carefully integrated, a server that decides to push its updates to a number of other servers simply uses a single multicast group to send its updates.
- In contrast, with a pull-based approach, it is generally only a single client or server that requests its copy to be updated.
- In that case, unicasting may be the most efficient solution.

Consistency Protocols

- So far, we have mainly concentrated on various consistency models and general design issues for consistency protocols.
- We concentrate on the actual implementation of consistency models by taking a look at several consistency protocols.
- A consistency protocol describes an implementation of a specific consistency model.
- We follow the discussion on consistency models by first taking a look at data-centric models, followed by protocols for client-centric models.

Primary-Based Protocols

- In the case of sequential consistency, it turns out that primary-based protocols prevail.
- In these protocols, each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x .
- A distinction can be made as to whether the primary is fixed at a remote server or if write operations can be carried out locally after moving the primary to the process where the write operation is initiated.

Primary-Based Protocols

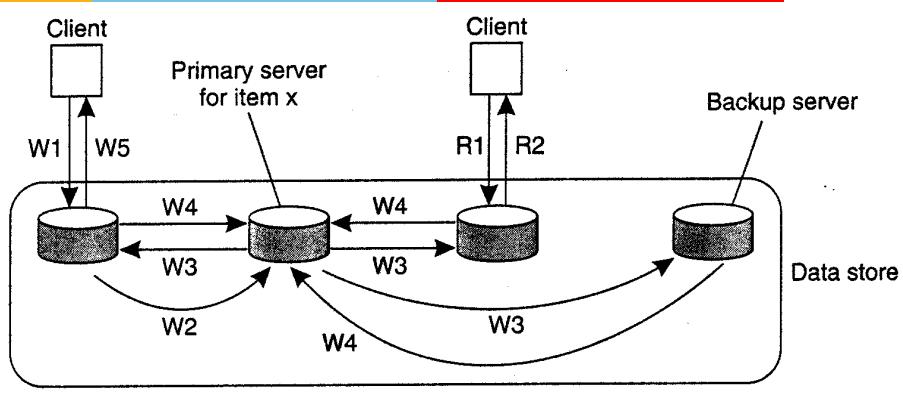
Remote- Write Protocols

- The simplest primary-based protocol that supports replication is the one in which all write operations need to be forwarded to a fixed single server.
- Read operations can be carried out locally.
- Such schemes are also known as primary backup protocols.

Primary-Based Protocols

- A process wanting to perform a write operation on data item x, forwards that operation to the primary server for x.
- The primary performs the update on its local copy of x, and subsequently forwards the update to the backup servers.
- Each backup server performs the update as well, and sends an acknowledgment back to the primary.
- When all backups have updated their local copy, the primary sends an acknowledgment back to the initial process.

Primary-Based Protocols



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

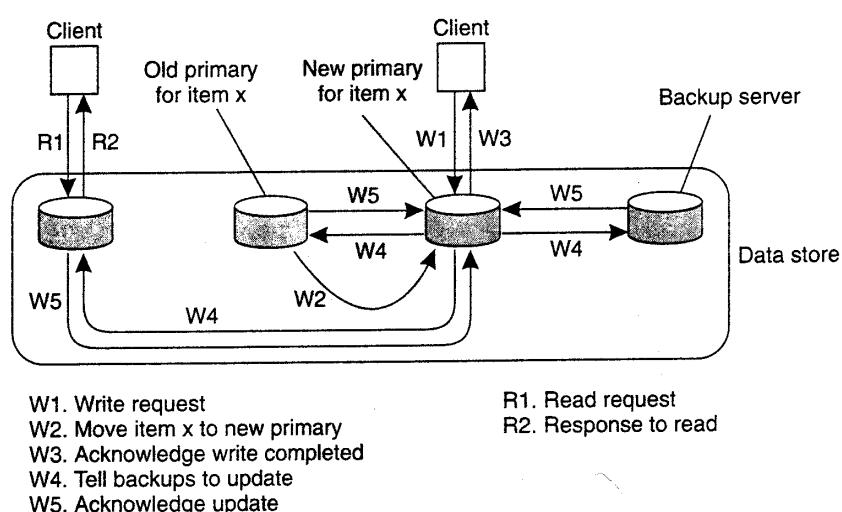
R1. Read request
R2. Response to read

Primary-Based Protocols

Local- Write Protocols

- A variant of primary-backup protocols is one in which the primary copy migrates between processes that wish to perform a write operation.
- As before, whenever a process wants to update data item x, it locates the primary copy of x, and subsequently moves it to its own location.
- The main advantage of this approach is that multiple, successive write operations can be carried out locally, while reading processes can still access their local copy.
- However, such an improvement can be achieved only if a nonblocking protocol is followed by which updates are propagated to the replicas after the primary has finished with locally performing the updates.

Primary-Based Protocols



Replicated-Write Protocols

- In replicated-write protocols, write operations can be carried out at multiple replicas instead of only one, as in the case of primary-based replicas.
- A distinction can be made between active replication, in which an operation is forwarded to all replicas, and consistency protocols based on majority voting.

Replicated-Write Protocols

Active Replication

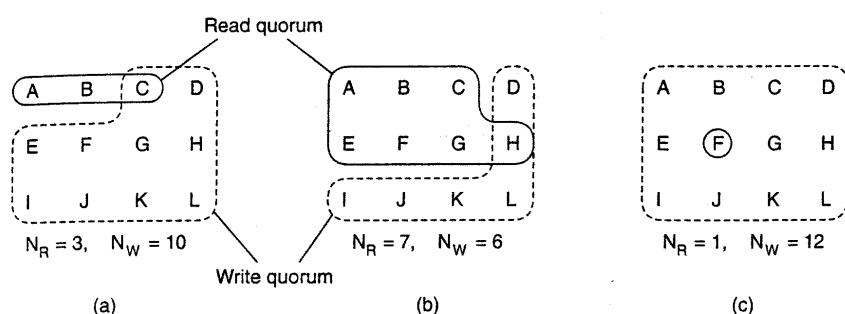
- In active replication, each replica has an associated process that carries out update operations.
- In contrast to other protocols, updates are generally propagated by means of the write operation that causes the update. In other words, the operation is sent to each replica.
- One problem with active replication is that operations need to be carried out in the same order everywhere.

Replicated-Write Protocols

Quorum-Based Protocols

- A different approach to supporting replicated writes is to use voting as originally proposed by Thomas (1979) and generalized by Gifford (1979).
- The basic idea is to require clients to request and acquire the permission of multiple servers before either reading or writing a replicated data item.

Replicated-Write Protocols



Three examples of the voting algorithm.

- (a) A correct choice of read and write set.
- (b) A choice that may lead to write-write conflicts.
- (c) A correct choice, known as ROW A (read one, write all).

Cache-Coherence Protocols

- Caches form a special case of replication, in the sense that they are generally controlled by clients instead of servers.
- However, cache-coherence protocols, which ensure that a cache is consistent with the server-initiated replicas are, in principle, not very different from the consistency protocols discussed so far.

THANK YOU

IMP Note to Self



STOP RECORDING