


# Introduction to Distributed System

The background of the slide features a blue gradient that transitions from a deep blue on the left to a lighter, cyan-like blue on the right. In the lower half of the image, there are several overlapping, wavy lines in shades of yellow, light blue, and white, creating a sense of movement and depth.

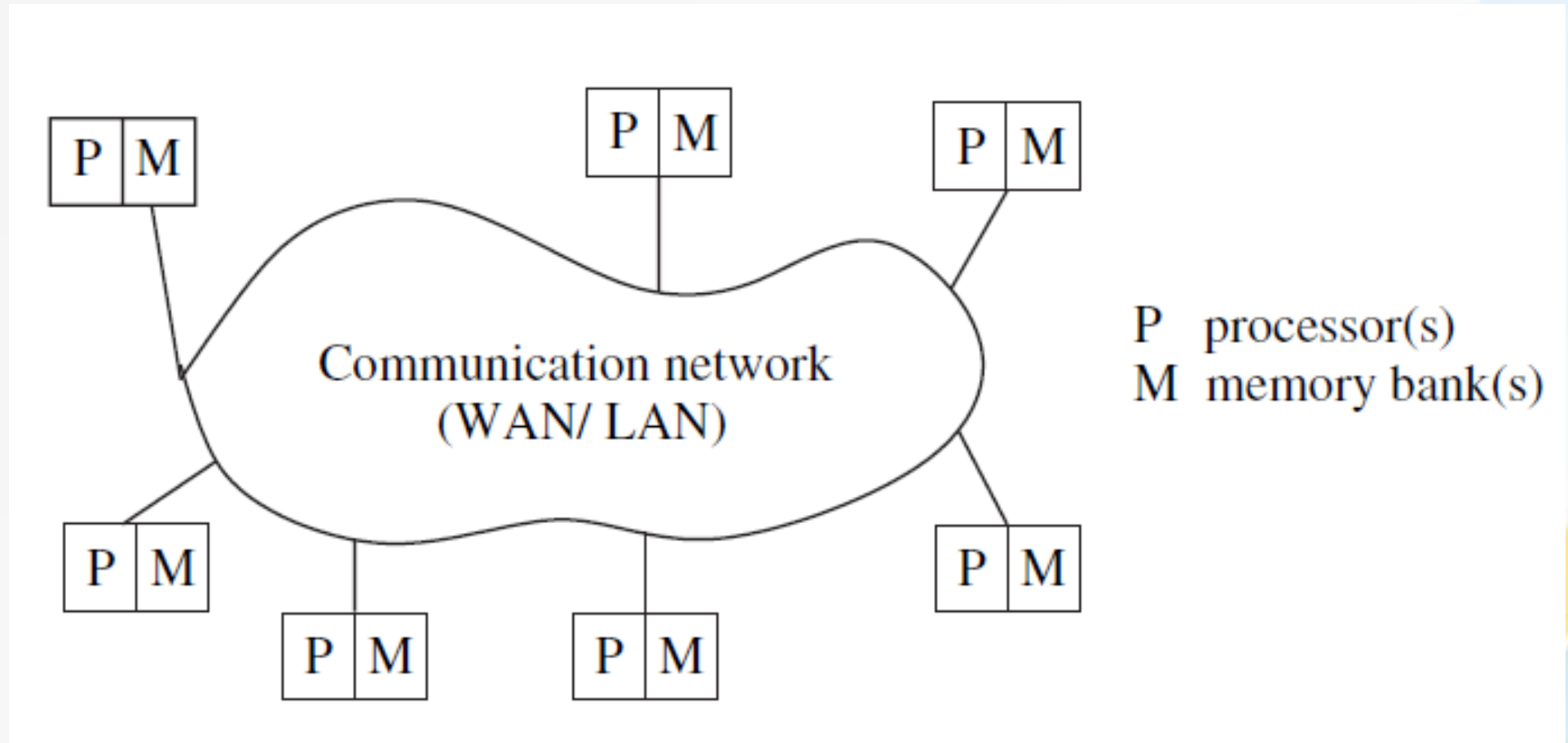
# Agenda

- Definition of a Distributed System
  - Typical Distributed System
  - Relation with Parallel Systems
  - Interaction with System Components
  - Motivation
- 
- A decorative graphic on the right side of the slide, consisting of several overlapping, curved, wavy shapes in shades of light blue, yellow, and a darker blue at the bottom right corner.

# Definition of a Distributed System

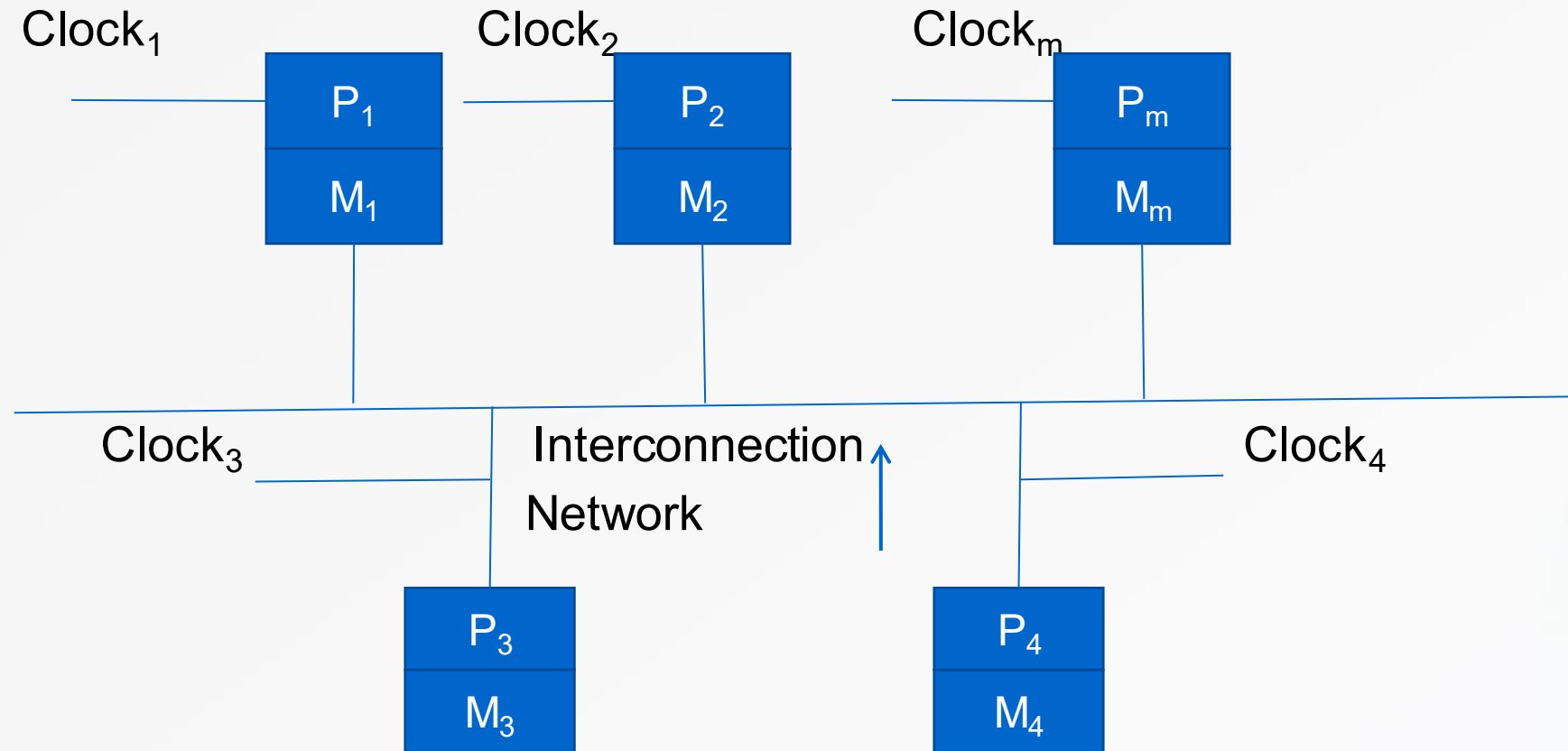
- A collection of independent computers that appears to the users of the system as a **single coherent computer/system image**.
- A collection of computers that do not **share common memory** or a **common physical clock**, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system.
- Distributed Operating System- Identical OS on all the nodes in the system.
- Nodes typically are **loosely coupled**.
- A distributed system is a **collection of independent entities** that cooperate to solve a problem that cannot be individually solved.  
(Not feasible to have a single node with huge computing power)

# Typical Distributed System



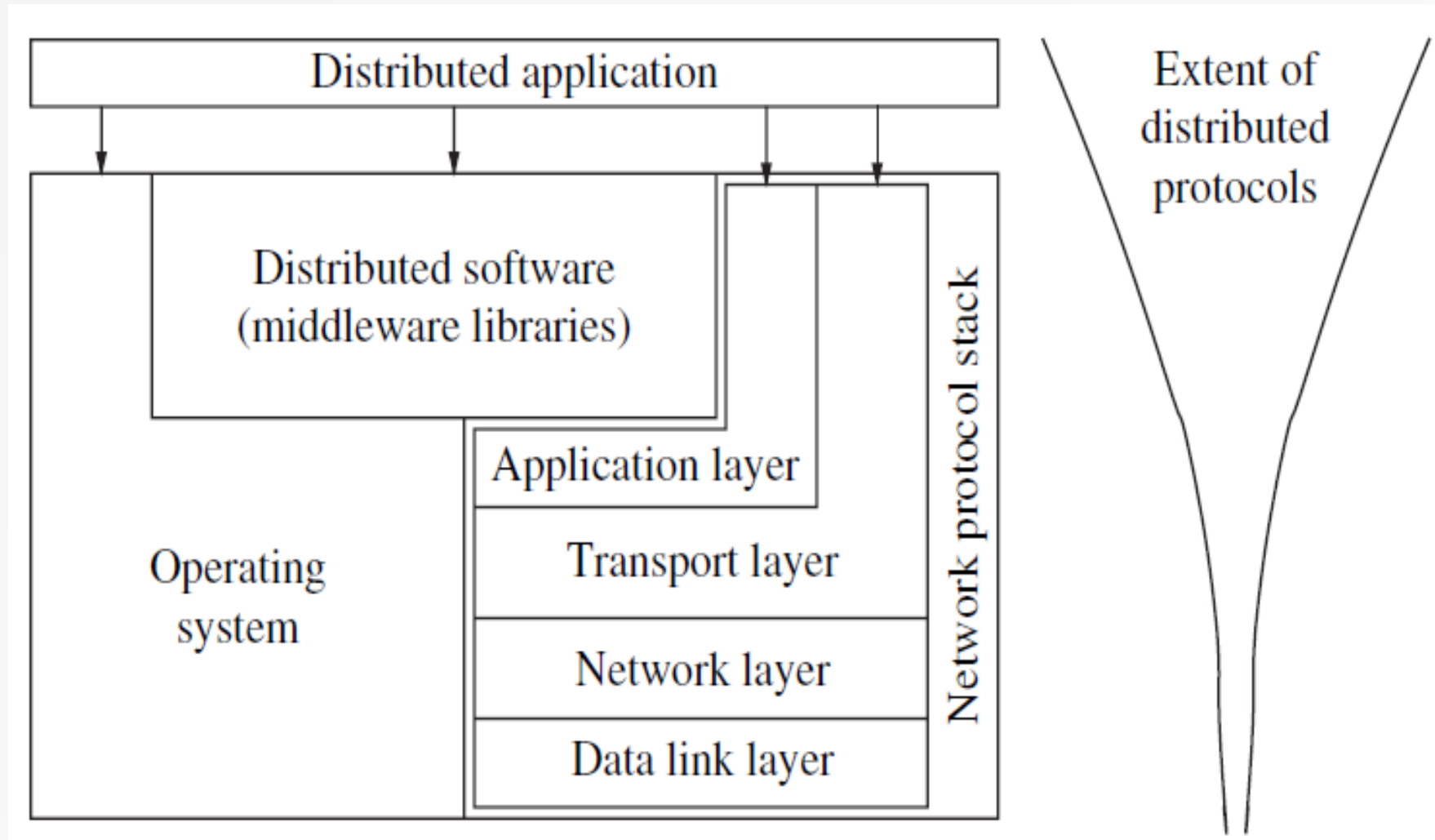
Ref: Distributed Computing: Principles, Algorithms, and Systems by  
Ajay D. Kshemkalyani and Mukesh Singhal

# Another Representation of Distributed Systems



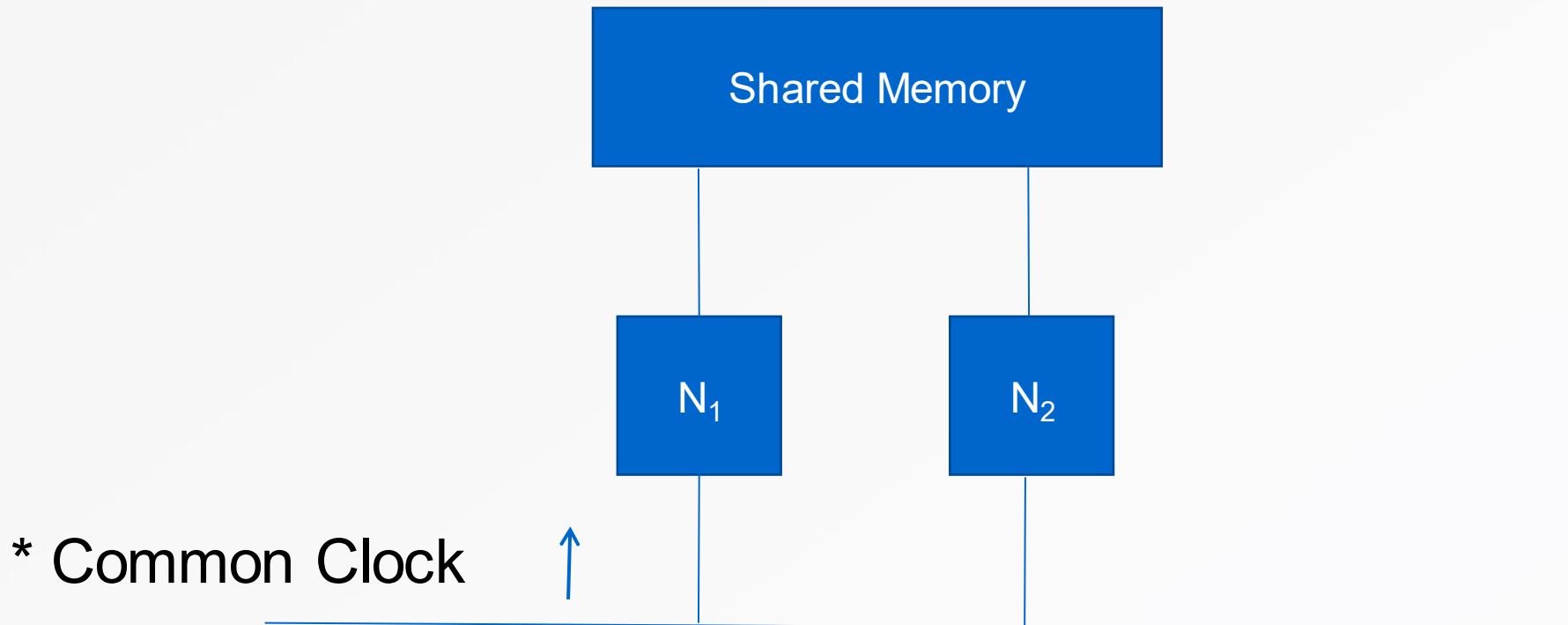
- Nodes may be geographically apart, therefore, there may be longer delays in communication.
- Clock at each of the nodes is different.

# Interaction of the Software Components at each Processor



# Multiprocessor/Parallel Systems

- A multiprocessor system is a parallel system in which the multiple processors have direct access to a shared memory which forms a common address space.
- 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.



# Motivation

## ➤ **Inherently distributed computations**

- In most of the applications, the computation is inherently distributed.
- For ex. Money transfer in banking.

## ➤ **Resource sharing**

- Full replication of resources such as printers, complete data sets in databases, files is not feasible at all places because it is not practical as well as not cost-effective.
- Placing these resources at a single place/site could be a bottleneck and therefore, these are typically distributed across the system.



# Motivation

- **Access to geographically remote data and resources**
  - The data replication may not possible at every site because it may be too large or too sensitive to be replicated.
  - Ex. Payroll data within a multinational company - large and sensitive to be replicated at every branch office/site.
  - Stored at a central server which can be queried by branch offices.
  - Supercomputers (special resources) exist only in certain locations
  - For accessing them, users need to log in remotely.

# Motivation Cont..

- **Solving big problems**
- Data is not possible to be available in one system and also the processing power on a single node also will not be enough to solve the big problems.
- Ex. Whether Forecasting, Google Search
- **Scalability:**
  - As the processors are usually connected by a wide-area network (WAN), more processors can be added without introducing any problem for the communication network.
  - The number of users can be added (can be reduced) on fly as well
  - Ex. Amazon sale for 3 hrs.

# Motivation Cont..

## ➤ **Modularity and incremental expandability**

- The addition of heterogeneous processors may be done to the system without affecting the performance assuming that those processors are running the same middleware algorithms.

# Challenges in DCS

The background of the slide features a gradient of blue colors, transitioning from a deep blue on the left to a lighter, cyan-like blue on the right. Overlaid on this gradient are several thick, wavy, horizontal bands. From top to bottom, these bands are colored orange, light blue, and white. The waves in the bands are smooth and flowing, creating a sense of movement and depth.

# Challenges in Distributed System

## ➤ **Heterogeneity**

- Interoperability between various heterogeneous components of the system
- Ex. Underlying Network, Operating System, Architecture, Programming Languages

## ➤ **Openness**

- Determines whether the system can be extended and reimplemented in various ways.
- Ex. Internet is an open system as the specifications of Internet Protocols are published in RFCs (Requests for Comments-IETF).
- If the system is open, the developers can add new features or replace sub-systems in the future.
- Key interfaces are **published**.
- Ex. APIs are exposed by Twitter and Facebook that allow developers to develop their own software interactively (e.g., AI, business tools, analytical tool APIs).

# Challenges in Distributed System Cont..

## ➤ Scalability

- System should work efficiently with an increasing number of users and the number of resources.
- It must be possible to add server computers to avoid the performance bottleneck if a single file server had to handle all file access requests
- Preventing software resources running out
  - Ex. Lack of scalability: the numbers used as Internet (IP) addresses
- Avoiding performance bottlenecks
  - DNS predecessor used to be kept on a server and used to be downloaded whenever needed by a computer.

# Challenges in Distributed System Cont..

## ➤ **Failure Handling**

➤ A process, node or a network might fail independent of each other.

## ➤ **Detecting failures**

➤ Checksums can be used to detect corrupted data in a message or a file.

## ➤ **Masking failures**

➤ Messages can be retransmitted when they fail to arrive.

## ➤ **Recovery from failures:**

➤ Design of software so that the state of permanent data can be recovered or 'rolled back' after a server has crashed

## ➤ **Redundancy:**

➤ Routes in the internet, Database replicas

# Challenges in Distributed System Cont..

## ➤ **Concurrency:**

- There can be multiple users present in a distributed system and this is a source of concurrent requests to its resources.
- Ex. A data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time by multiple clients.
- All the resources must be designed to be safe in a concurrent environment.



# Challenges in Distributed System Cont..

## ➤ **Transparency**

- It is also called as abstraction in DS.
- Transparency deals with hiding the implementation policies from the user.
- There are various types of Transparency mechanisms in DS.

# Access Transparency

- Hide differences in the access of distribution of resources.
- The set of **operations (system calls)** to access the resources should remain same, but user is unaware of it (local or remote).
- Ex. Files in NFS, Navigation of Web Pages, etc.

# Location Transparency

- The location of a resource is hidden from the user.
- Ex. Files or groups of files may be relocated without changing their pathnames.
- Ex. Files in NFS, Navigation of Web Pages, etc.

# Migration Transparency

- The resource/process may be moved to another location.
- This is hidden from the user.
- This is done without affecting the operations of the users and the applications that are running.
- Used for load balancing by DS

# Replication Transperancy

- A resource may be replicated in several places and users are unaware of it.
- The data is replicated in two or more sites for more reliability.

# Transparency Cont..

## ➤ **Concurrency**

- Various users and applications should be able to access shared data or objects without interference between each other

## ➤ **Failure**

- Hide the failure and recovery of a resource
- Despite any failures in the system the task needs to be completed successfully.

# Challenges Cont..

## ➤ **Communication**

- Need to design mechanisms for inter-process communication in the network.

## ➤ **Examples**

- Remote Procedure Calls
- Remote Method Invocation

# Challenges Cont..

## ➤ **Mobile Code**

- Code is stored at a web server, client gets the code downloaded from a web server. The code runs at a client side, i.e., web browser.
- Ex. Java Applets
- Push model-server instead of the client initiates interactions (Reduction in delays from the server coz of N/W).
- Ex: a stockbroker might provide a customized service to notify customers of changes in the prices of shares.



# Challenges Cont..

## ➤ **Mobile Agent**

- A running program (both code and data) that goes from one node to another in a network doing a task on someone's behalf, such as collecting information, and finally returning with the results.
- Ex. compare the prices of products from a number of vendors by visiting each vendor's site and performing a series of database operations

# Challenges from System Perspective Cont..

## ➤ **Synchronization**

- Mutual exclusion
- Leader election
- Synchronizing physical clocks

## ➤ **Security**

- Various aspects of cryptography, secure channels, access control (authentication and authorization), key management - generation and distribution.

# Fundamental Models for Distributed Algorithms



# Outline

- Absence of a Global Clock
- Fundamental Models of Distributed Algorithms

# Absence of a Global Clock

- The notion of global time does not exist.
- Solutions
- Having a common clock to all the computers (processes) in the system.
  - Two different processes can observe a global clock value at different instants due to unpredictable message transmission delays.
- Having synchronized clocks at each computer
  - Each computer has a physical clock and synchronize these clocks among the nodes.
  - Because of the technological limitations, the clocks drift from the physical time.

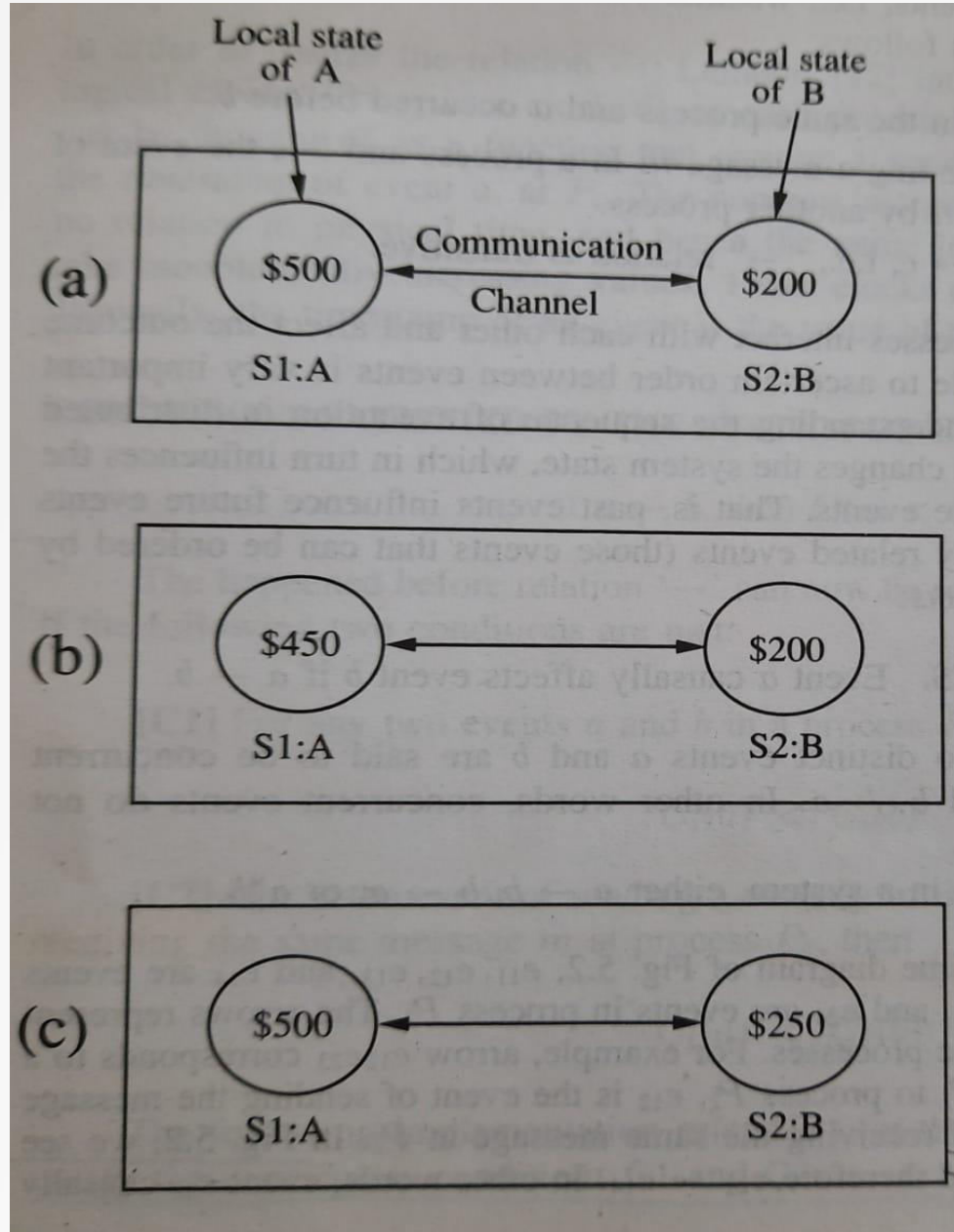
# Impact of the Absence of Global Time

- Important to have a temporal ordering in DS
- Ex. An Operating System schedules the processes
- Basic criterion in scheduling the processes is the temporal order in which requests to execute processes that arrive.
- Due to the absence of a global clock, it is difficult to reason about the temporal order of events in a distributed system.
- Difficult to collect up-to-date information on the state of entire system.

# Absence of Shared Memory/Global Clock

- Up-to-date global system state is required for debugging, recovering from failures, etc.
- Coherent View:
  - If all the observations of different processes are made at the same physical time.
- A global state or a complete view- local views (local state) at each of the nodes and messages that are in transit in DS.
- Difficult to obtain a coherent global state of the system.

# Example Demonstrating the Problem with a Global State



➤ Ex. S1 and S2 are two distinct sites that maintain bank accounts A and B.

➤ We would like to record a global state of banking transactions.

a. Initial state of the system

b. S1 records the system state after sending an amount to S2 and S2 records the system state before it receives an amount from S1.

c. S1 records the system state before sending an amount to S2 and S2 records the system state after S2 receives an amount from S1.

**\*Note:** The communication channel cannot record its state by itself.



# Fundamental Models (Assumptions) of Distributed Algorithms

- Network Topology used
- Communication Medium
- Synchronous or Asynchronous System
- Failures of Nodes

# Network Topology Used

- Nodes running the distributed algorithms will assume the following topologies (logical).
- Completely connected, Ring, Tree, etc.
- The underlying topology (physical) may be anything else than stated above.

# Communication Medium

- Message Passing
- Reliable (TCP) or Unreliable (UDP- No guarantee)
- Ordering of messages
  - In-order delivery (TCP)
  - If the messages are out of order, then TCP guarantees that the messages will be made in-ordered before giving to an application.
  - Out-of order delivery (UDP)



# Communication Medium Cont..

- Message latency

- TCP (more latency) /UDP (less latency)

- Unicast/Multicast/Broadcast Support

- Spanning tree over the graph (Network modeled as a graph)

# Synchronous or Asynchronous System

## ➤ Synchronous System

- In a synchronous system, there is a bound on the time or message latency.
- A blocking type model wherein sender process gets blocked until a receiver process receives the message.
- Sender process resumes an execution only after it comes to know that the receiver has accepted the message.
- If the message does not reach within the bound assumed by the system, message will be retransmitted.

## ➤ Asynchronous System

- There is no bound on the message latency.
- Non-blocking, buffering mechanism at receiver end.
- DCS are inherently asynchronous

# Failures of Nodes

- Specifies the manner in which the component(s) of the system may fail.
- **Fail Stop**
  - A properly functioning process may fail by stopping execution from some instant.
  - Other processes can detect certainly that the process has crashed (using a timeout mechanism).
- **Crash Failures**
  - A properly functioning process may fail by stopping execution from some instant.
  - If nodes crash, so there is no computation or no communication, i.e. other processes do not learn of this crash ( Assumed to be asynchronous system, a timeout can indicate only that a process is not responding).

# Failures of Nodes

## ➤ **Crash Failures Cont..**

- A timeout can indicate only that a process is not responding - it may have crashed or may be slow, or the messages may not have arrived.

## ➤ **Arbitrary Failures**

- Alter the contents of messages -Worst case
- It might corrupt the messages intentionally (security attack)

# Performance Measures for Distributed Algorithms

The background of the slide features a blue gradient that transitions from a deep blue on the left to a lighter cyan on the right. Overlaid on this gradient are several wavy, horizontal bands of color. These bands include shades of light blue, a prominent yellow-orange, and a pale cream or off-white at the bottom. The lines are smooth and fluid, creating a modern, abstract aesthetic.



# Agenda

- Performance Measures for Distributed Algorithms
  - Message Complexity
  - Bit Complexity
  - Time Complexity
  - Space Complexity
- Application Demonstration

# Message Complexity

## ➤ Centralized Algorithms

- Time complexity and Space complexity

## ➤ Distributed Algorithms

- It is a measure of number of messages sent
- Time is not considered
- Network may or may not be congested
- Clocks at nodes are not synchronized

# Bit Complexity and Time Complexity

## ➤ Bit Complexity

- Number of bits transmitted
- Messages are in general short but are of variable sizes
- It can fit in one packet

## ➤ Time Complexity

- It can be computed only when the system is synchronous
- Distributed systems are inherently asynchronous

# Space Complexity

- Storage required for algorithm
- Storage required for inputs taken for the algorithm
- Total number of bytes required by all the nodes in the system.

# Application - Distributed Search

- Distributed System is modeled as a graph  $G$  of  $n$  nodes and  $m$  edges.
- Assumptions
  - Nodes know only their neighbours
  - All the nodes are distinct, i.e., they have unique ids.
- Application needs to be evaluated using various fundamental models and its performance is to be measured using the measures mentioned.

# Model 1

- Asynchronous system
- Topology: Completely connected
- Communication: Reliable
- No node failures

# Algorithm

## ➤ Query Node


1. Sends a query to search an item to all the nodes (Msg Send)
2. If any node replies "Found"  
    return node\_id  
Else  
    return "Not Found" (Msg Recv)

## ➤ Other Nodes

1. On receiving/receipt of a query message, the node searches for an item. (Msg Recv)
2. Return "Found" / "Not Found" (Msg Send)

Message Complexity:  $2(n-1)$  :  $(n-1)$  queries and  $(n-1)$  replies.

# Model 2

- Asynchronous system
  - Topology: Completely connected
  - Communication: Unreliable
  - No node failures
- 
- A decorative graphic on the right side of the slide, consisting of several overlapping, curved, wavy shapes in shades of light blue, yellow, and dark blue, creating a modern, abstract background element.



# Algorithm

## ➤ Query Node

1. Sends a query to search an item to all the nodes (Msg Send)
2. If any node replies "Found"  
return node\_id  
If "Time Out"  
re-send query message to the nodes those did not reply.  
goto step 2
3. Return "Not Found"  
(Msg Recv)

\* (Execute (2) 'k' times) value of timeout?

## ➤ Other Nodes

1. On receiving/receipt of a query message, the node searches for an item. (Msg Recv)
2. Return "Found" / "Not Found"  
(Msg Send)

Message Complexity:

Best case:  $2(n-1)$

Worst case:  $2k(n-1)$

# Model 3

- Topology: Completely connected
- Communication: Reliable
- No node failures
- Synchronous system
  - There is a bound on the message latency
- Assumptions for Synchronous system
  - Bound on message latency =  $L$
  - Max search time at a node =  $S$

# Algorithm

## ➤ Query Node

1. Sends a query to search an item to all the nodes (Msg Send)
  2. Wait for  $2L + S$  time
  3. If any node replies "Found"  
return node\_id
- Else  
return "Not Found"  
(Msg Recv)

➤ Time Complexity:  $2L + S$

## ➤ Other Nodes

1. On receiving/receipt of a query message, the node searches for an item locally. (Msg Recv)
2. If an item is found  
Reply "Found" to the query node  
(Msg Send).

## ➤ Message Complexity

1.  $(n-1) + k$  - if  $k$  replies Yes.
2.  $(n-1) + 1 = n$  - if only 1 reply
3.  $(n-1)$  if there are no replies

# Distributed Communication

The background features a gradient from dark blue at the top to light blue at the bottom. Overlaid on this are several wavy, horizontal bands of color: a light blue band, a yellow band, and a white band at the bottom. The text 'Distributed Communication' is centered in the upper half of the image.

# Agenda

- Sockets

- Types of Sockets

- Connection-oriented Sockets
  - Connectionless Sockets

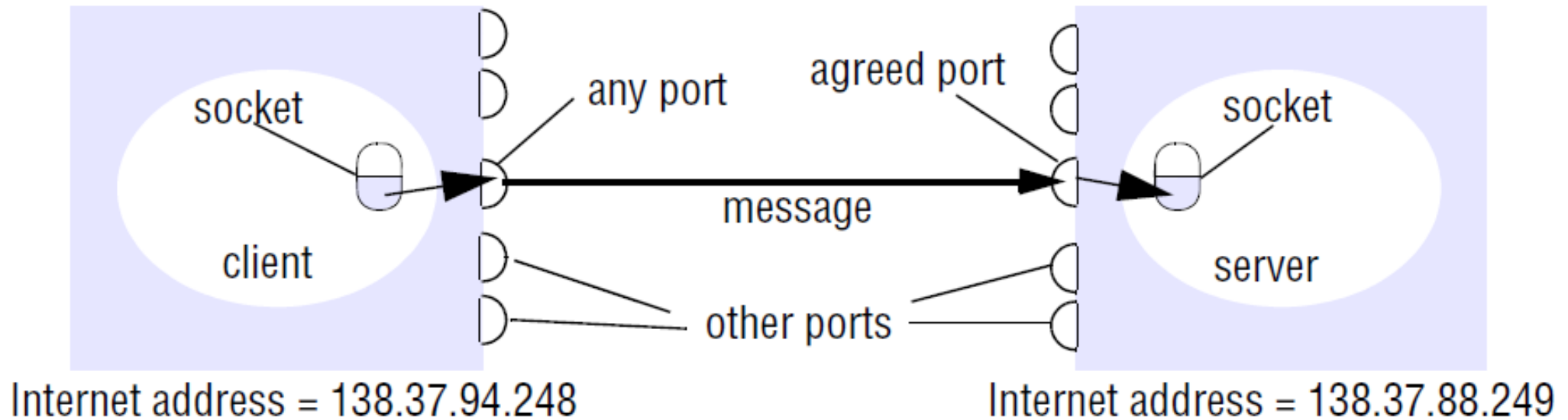
# Sockets

- Socket is a message passing abstraction and is an endpoint that you can name and address in a network (IP address and port number).
- Commonly used for the communication between any two hosts on the network.
- It is a lower level implementation of message passing in the distributed system.

## Sockets cont..

- All high-level programming mechanisms (RPC, RMIs, etc.) use sockets ultimately to communicate through the network.
- Socket application programming interfaces (APIs) are the network standard for TCP/IP.
- Most operating systems support socket APIs.

# Sockets and Ports



Reference: Distributed Systems - Concepts and Design  
(Fifth Edition) by George Coulouris, et. al.



# Types of Sockets

- Two types of sockets
- Connection-oriented sockets (streams)
- Connectionless sockets (datagram)

# Stream Socket

- Dedicated & end-to-end channel between server and client.
- Use TCP protocol for data transmission.
- Reliable and Lossless.
- Data sent/received in the same order.
- Long time for recovering lost/mistaken data

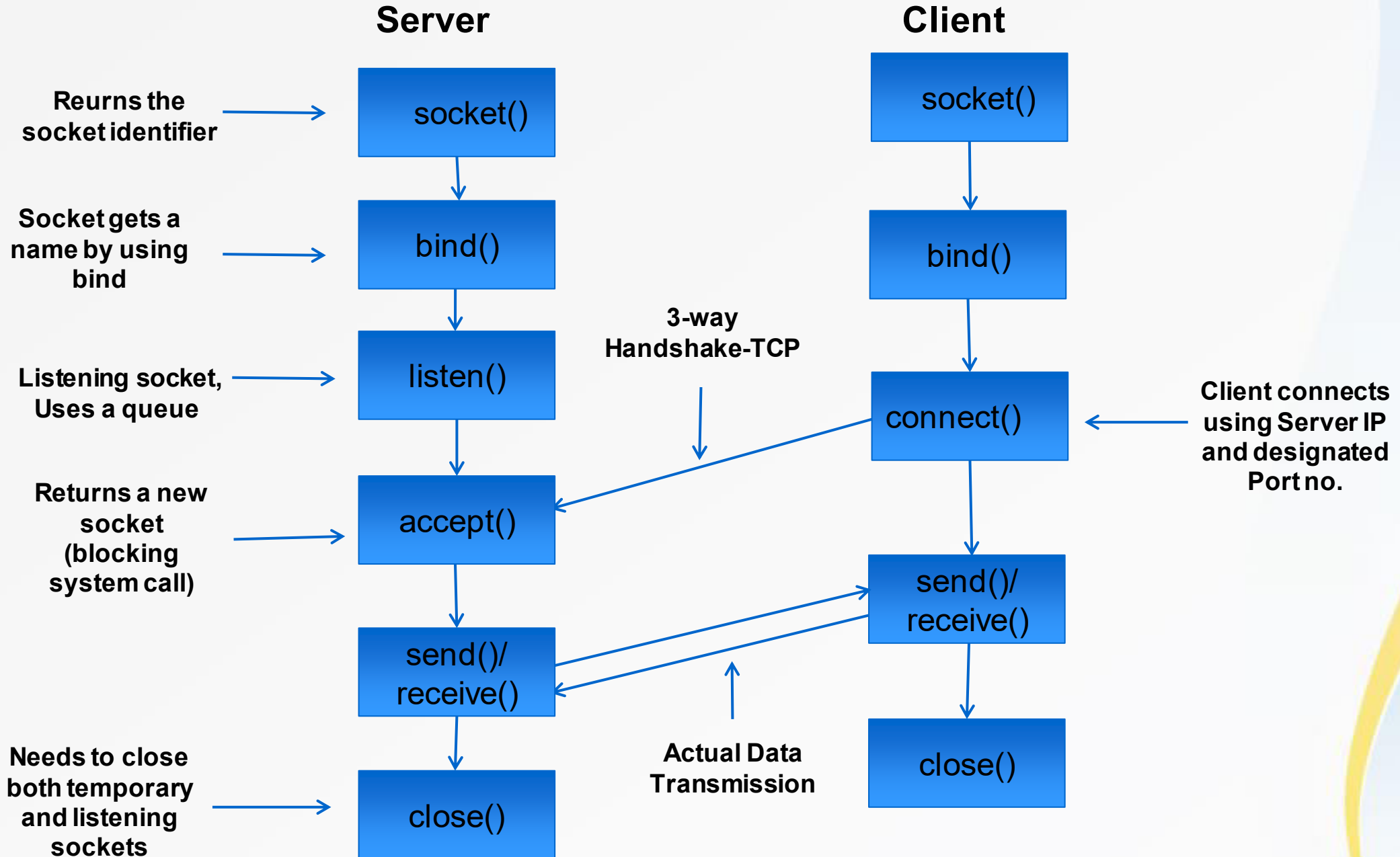
# Connection-oriented Sockets

- Connection-oriented sockets use stream.
- In a connection-oriented client-to-server model, the server side socket on the process waits for requests from a client.
- The server initially establishes an address that clients can use to find the server.

## Connection-oriented Sockets cont..

- After the address is established, the server waits for clients to request a service.
- The data exchange between client-to-server takes place when a client connects to the server through a socket.
- The server accepts the client's request, perform and sends the reply back to the client.

# Connection-oriented Sockets cont..



# Connection-oriented Sockets cont..

- The `socket()` API is used to create an endpoint for communications and returns a socket descriptor that represents the endpoint.
- Servers will bind a name to be accessible from the network and clients too bind a name to an IP address and port.
- The `listen()` API called by server to accept client connection requests.
- The `listen()` API is called after a socket is allocated with a `socket()` API and the `bind()` API binds a name to the socket.
- The `listen()` API must be called before an `accept()` API is called.

# Connection-oriented Sockets cont..

- The client application calls a `connect()` API on a socket to establish a connection to the server (3-way handshake of TCP).
- The server uses the `accept()` API to accept a client connection request.
- Once the connection is formed between client and server sockets, the calls like `send()`, `recv()`, `read()`, `write()`, can be used to initiate data transfer.
- A server or client issues a `close()` API to stop the data transfer.

# Protocol Control Block at Server

	Local Address	Local Port	Remote Address	Remote Port
Listening Socket -Listen()	Server IP Address	Server port		
Temporary Socket 1 accept()	Server IP Address	Server port	Client 1 IP Address	Client 1 port no.
	Server IP Address	Server port	Client 2 IP Address	Client 2 port no.
Temporary Socket 2 accept()	.			
	.			
	.			
Temporary Socket N accept()	Server IP Address	Server port	Client N IP Address	Client N port no.



# Protocol Control Block at Client

**Connect()** →

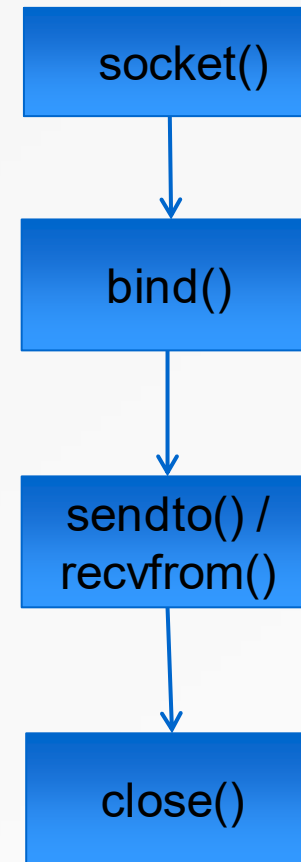
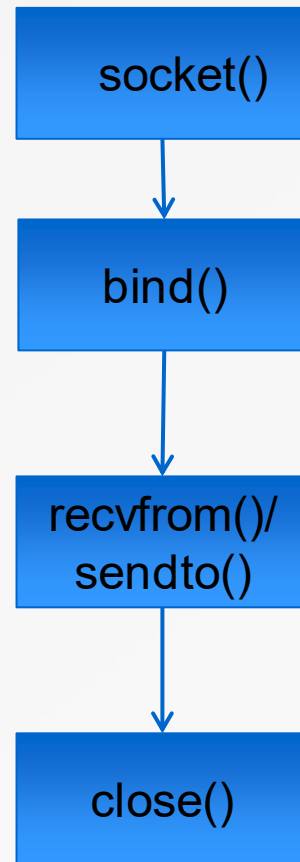
Local Address	Local Port	Remote Address	Remote Port
Client IP Address	Client Port No.	Server IP Address	Server port

Client sends its own IP address as well as port number to Server.

# Datagram Socket

- Not dedicated & end-to-end channel between server and client.
- Use UDP for data transmission.
- Not 100% reliable and may lose data.
- Data sent/received order might not be the same.
- Don't care or rapid recovering lost/mistaken data.

# Connectionless Sockets





# Distributed Communication - Remote Procedure Calls

# Introduction

- Socket mechanism is a lower-level communication that uses read-write /send-recieve for communication.
- Can we have some way of doing distributed computing that look like traditional centralized computing?
- Remote procedure call is a procedure call made to a procedure that is defined remotely on a particular host.

# Introduction

- 1984: Birrell & Nelson
- Goal : When a user calls a remote procedure, it should look like a local procedure called by the user.
- Procedure calling mechanism.

# How does a procedure call work?

- Machine instructions for call and return
- Compiler makes the calls abstracted.
- Passing parameters
- Local variables
- Return value.

# Regular Procedure Calls

- For a function,  $y = \text{fun}(a, b, c)$ 
  - The compiler will parse and generate the code to
  - push these parameters and generate a call to fun
- In compiling “fun”
  - Push the registers to save the values
  - Store local and temporary variables on the stack
  - Before a return, deallocate the stack, put the return data in a register, and issue a return instruction



# Remote Procedure Calls

// Client side	// Server side
main( )	fun( )
{	{
fun();	...
}	}

## ➤ How to support?

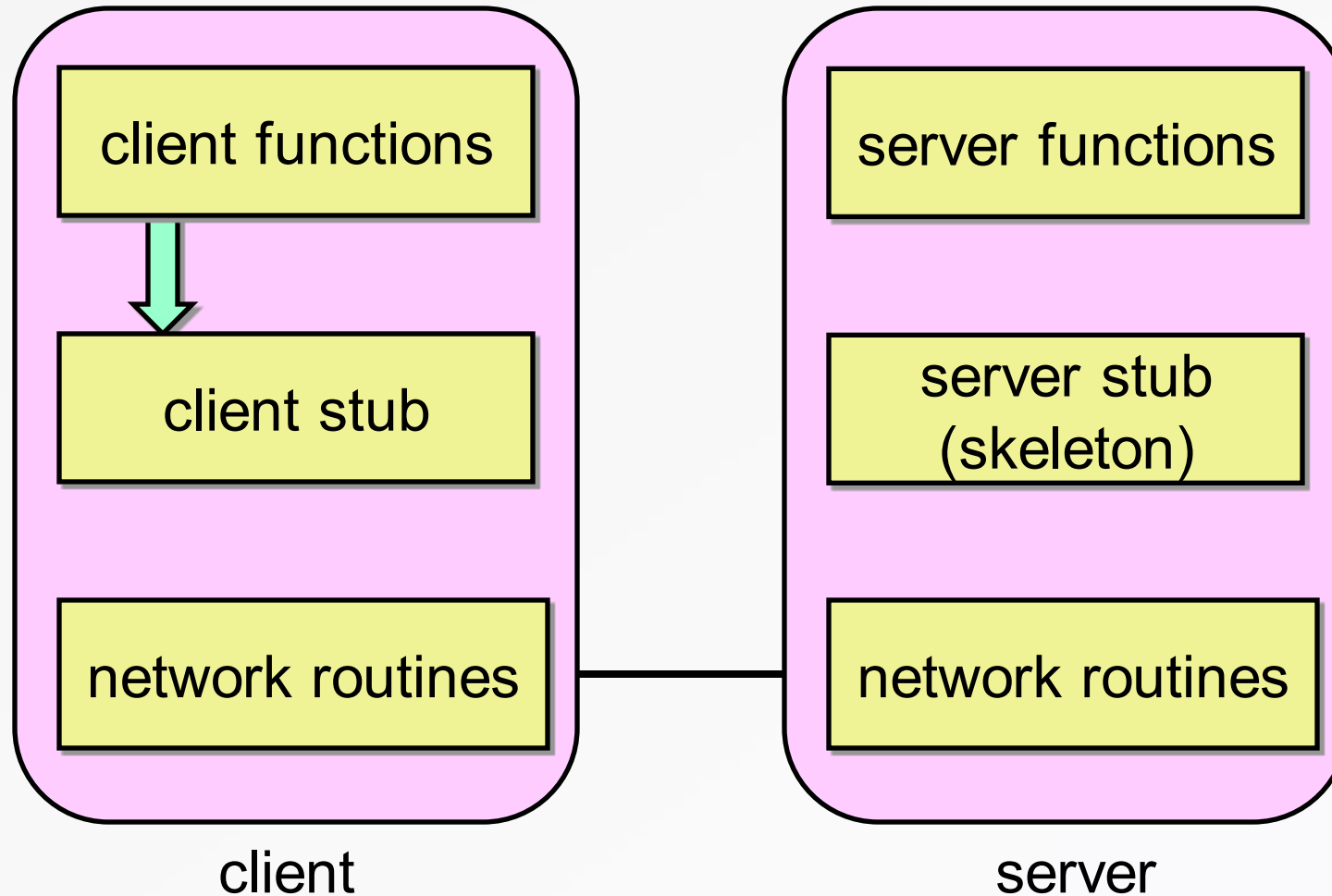
- Message passing - sockets
- Messages of parameters and returns are passed using sockets
- No architectural support for remote procedure calls
- Language-level construct

# How do we support RPC?

- Server puts the signature of functions on central repository
- Fundamental Idea:
  - Server process exports an *interface* of functions that can be called by client programs
- Clients make local procedure/function calls
- The function call is converted into a message exchange with remote server process
- Building messages, formatting, uniform representation, etc., is hidden in the stubs.

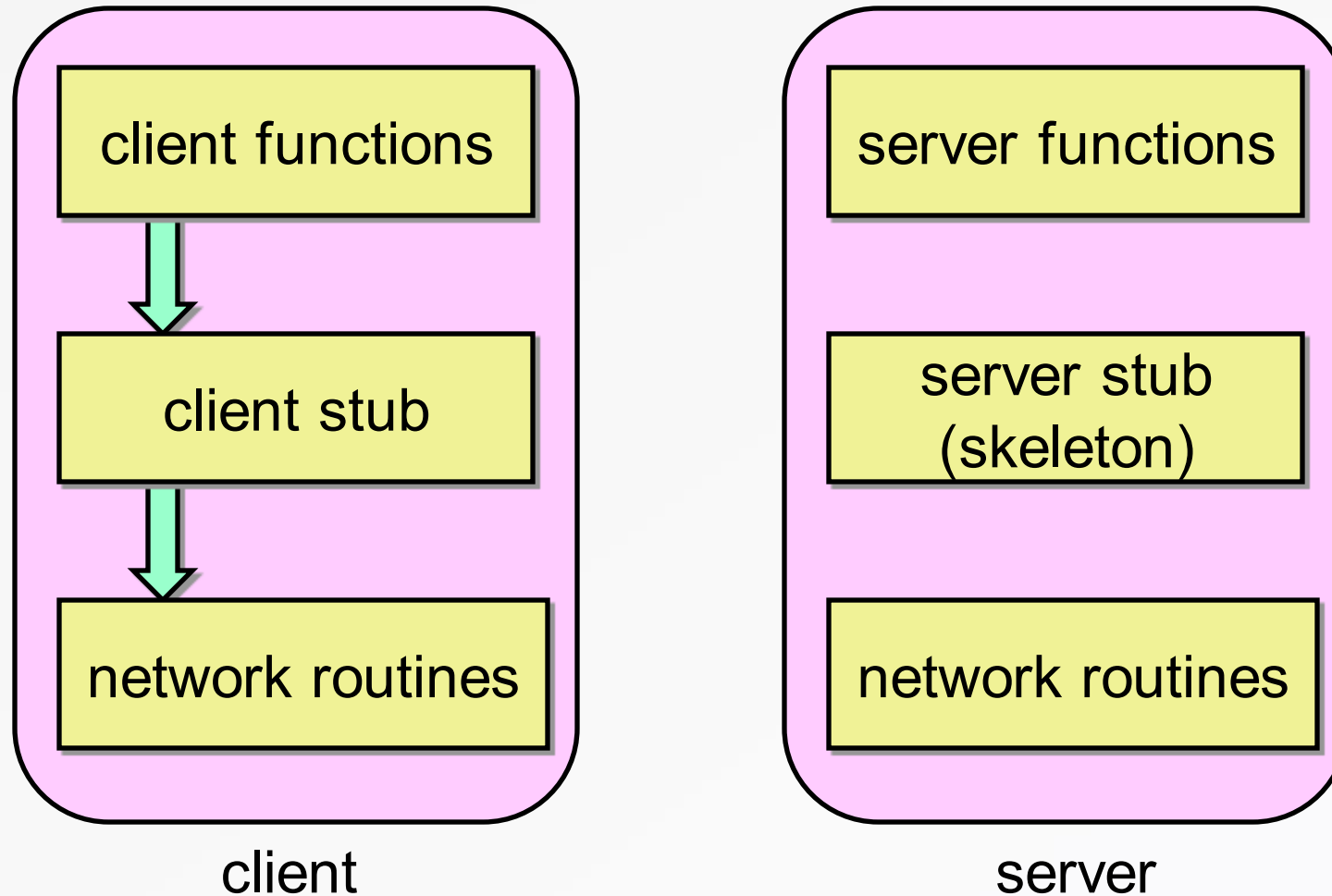
# Stub functions

## 1. Client calls stub (params on stack)



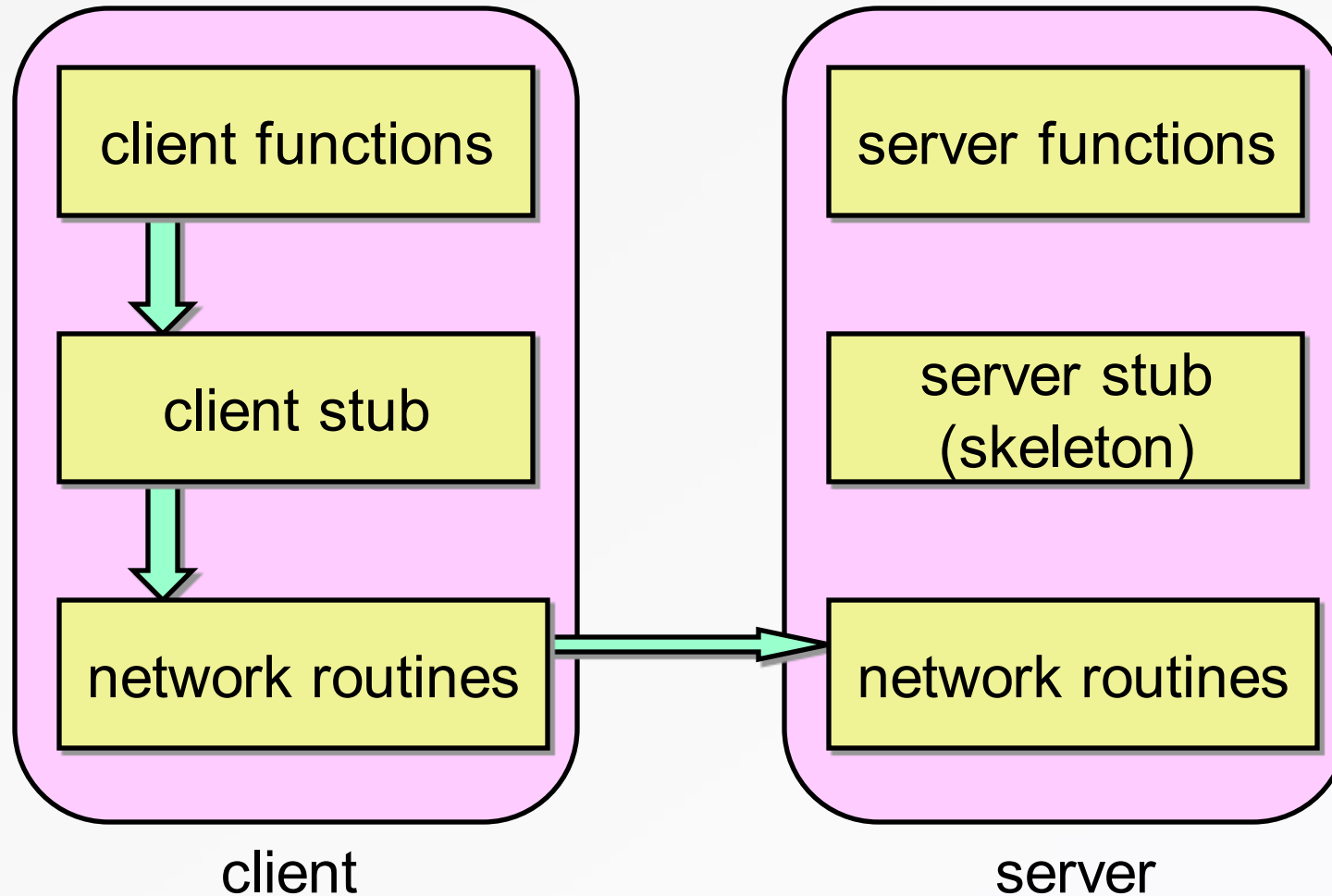
# Stub functions

## 2. Stub marshals params to a network message



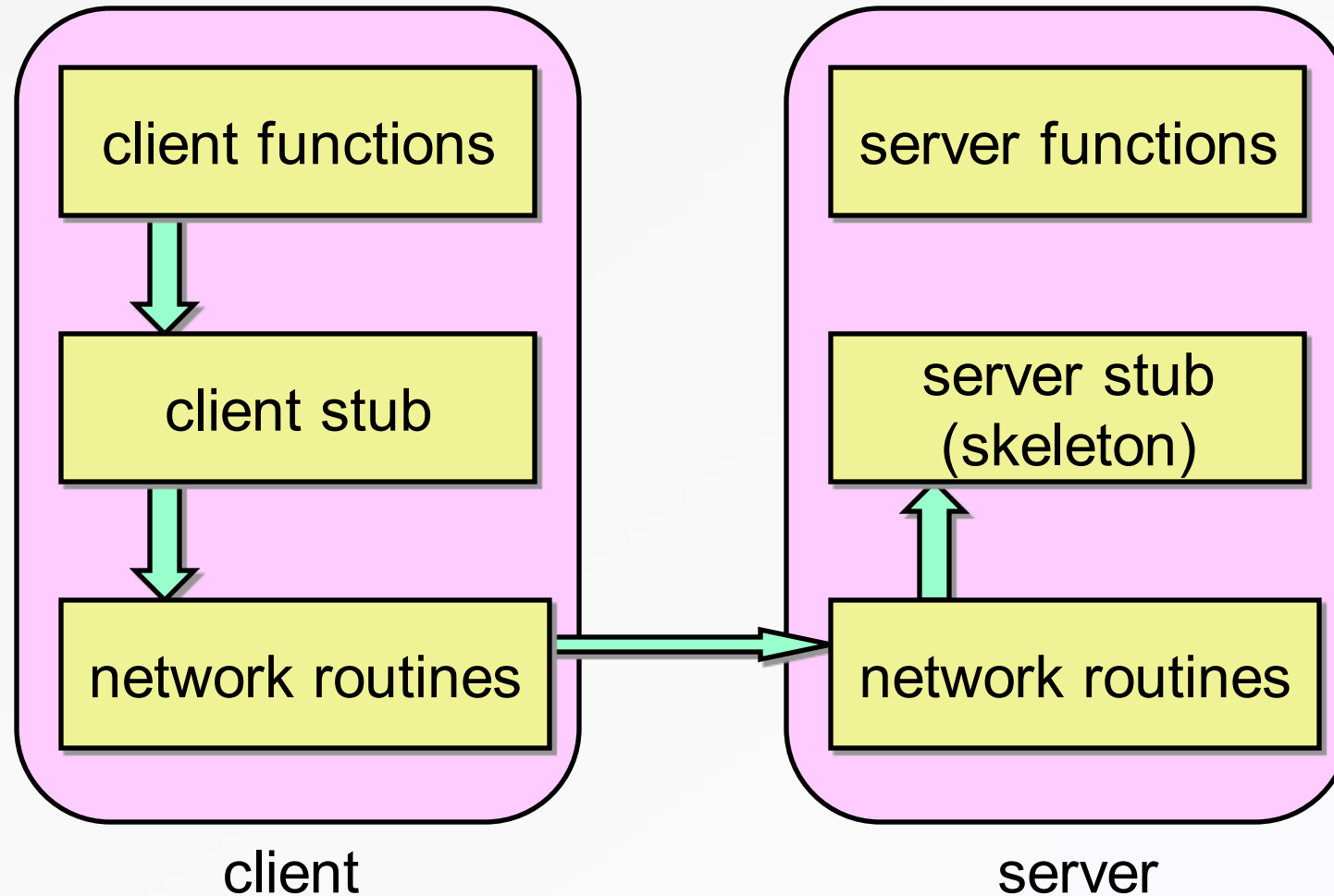
# Stub functions

## 3. Network message sent to server



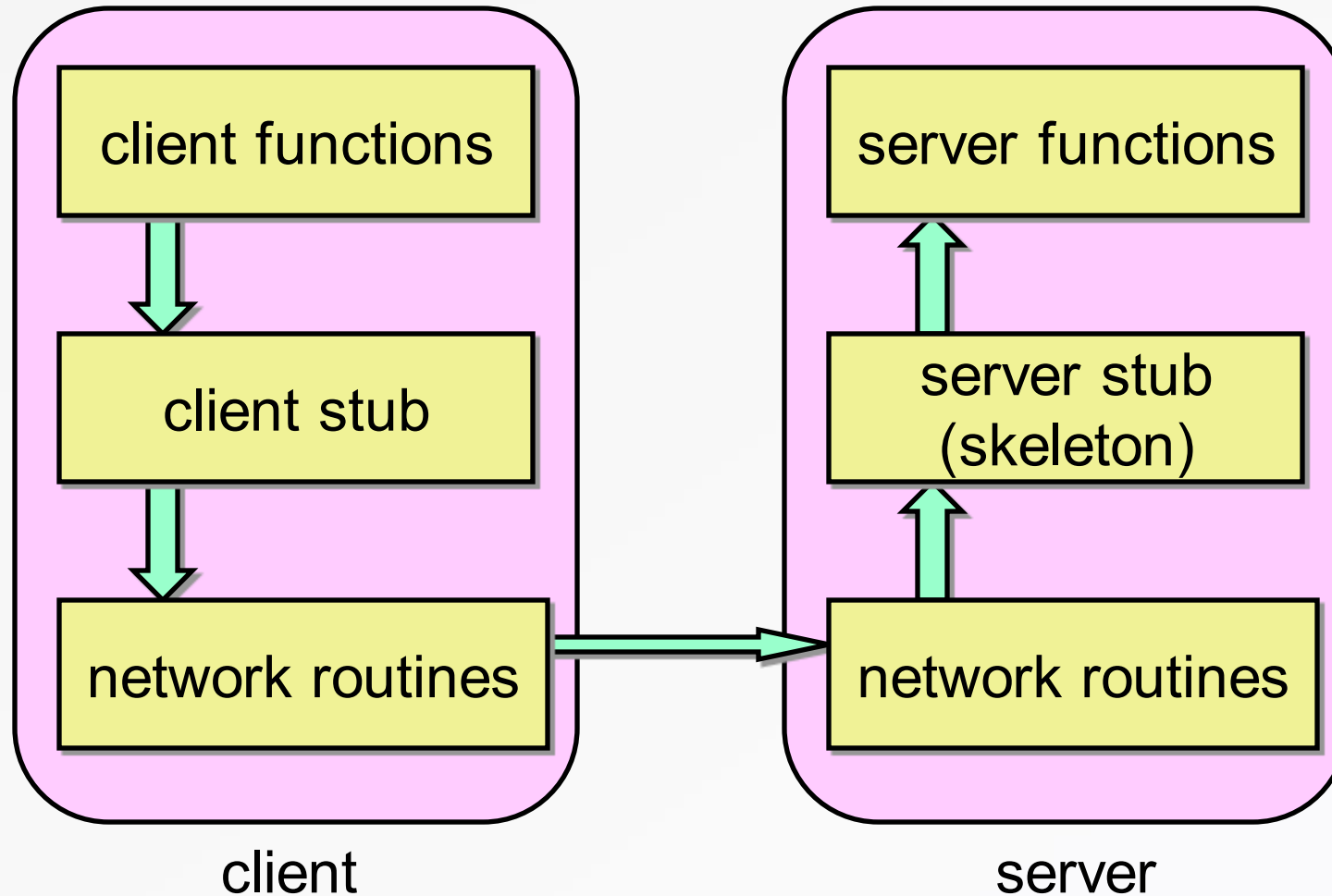
# Stub functions

## 4. Receive message: send to stub



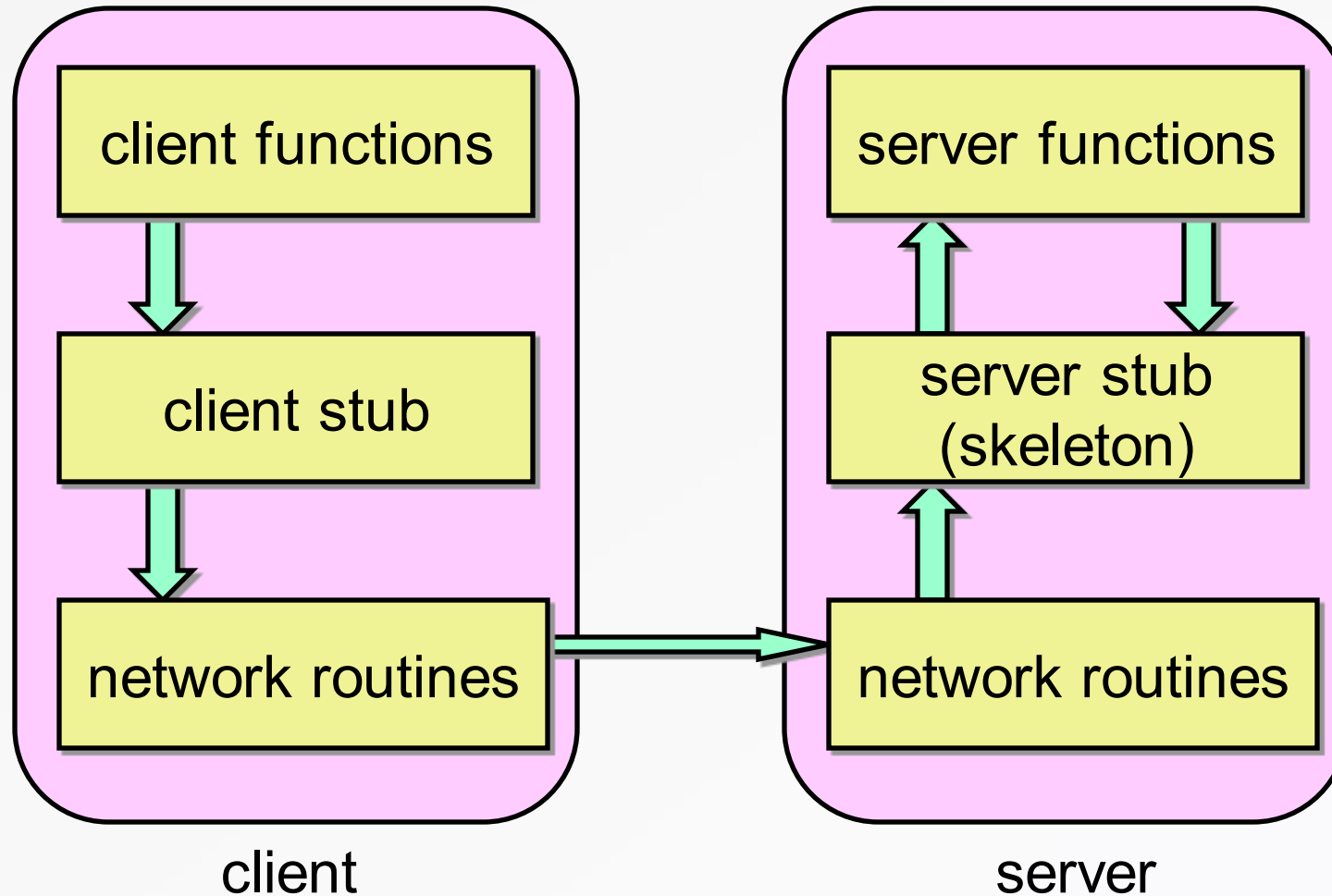
# Stub functions

## 5. Unmarshal parameters, call server func



# Stub functions

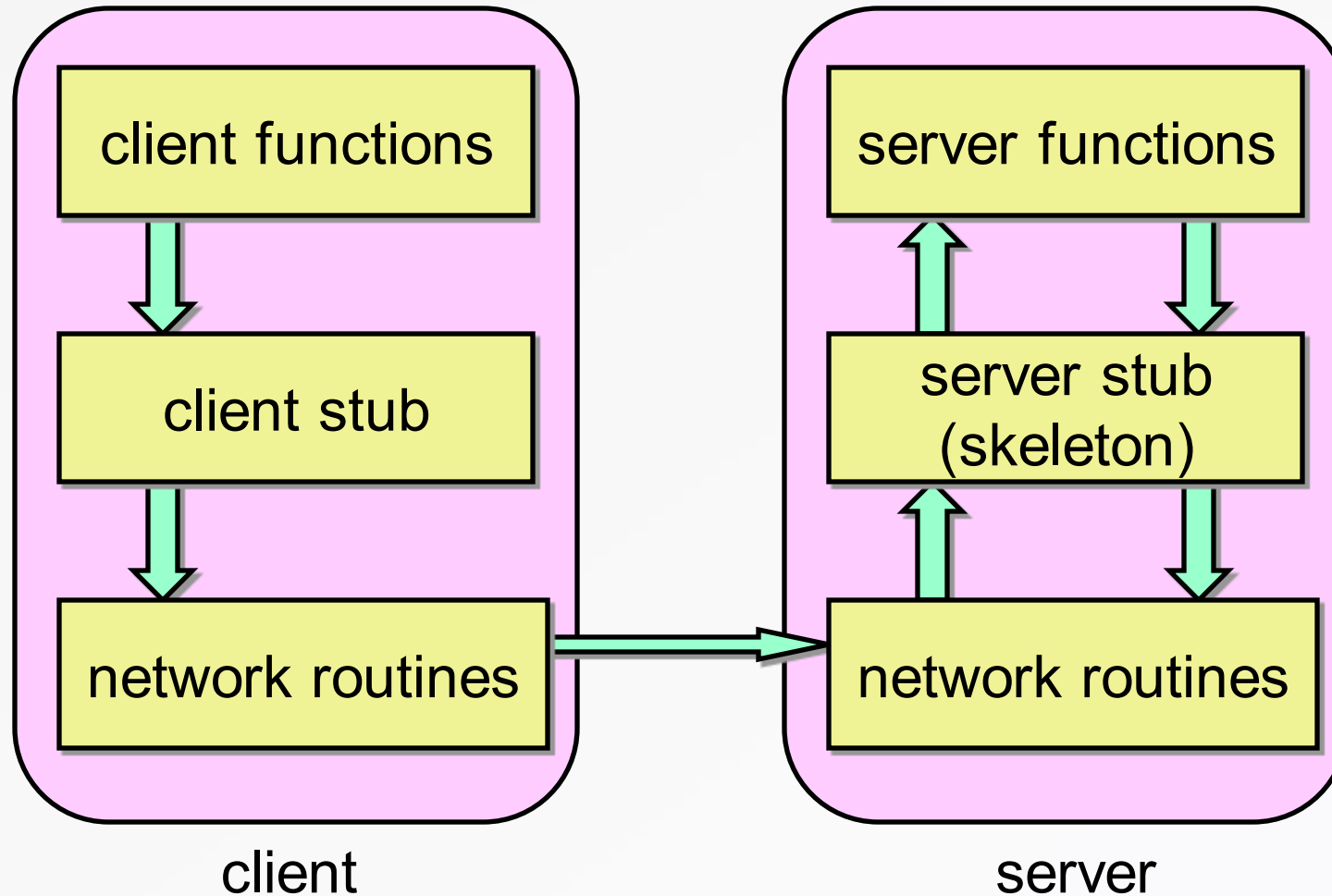
## 6. Return from server function





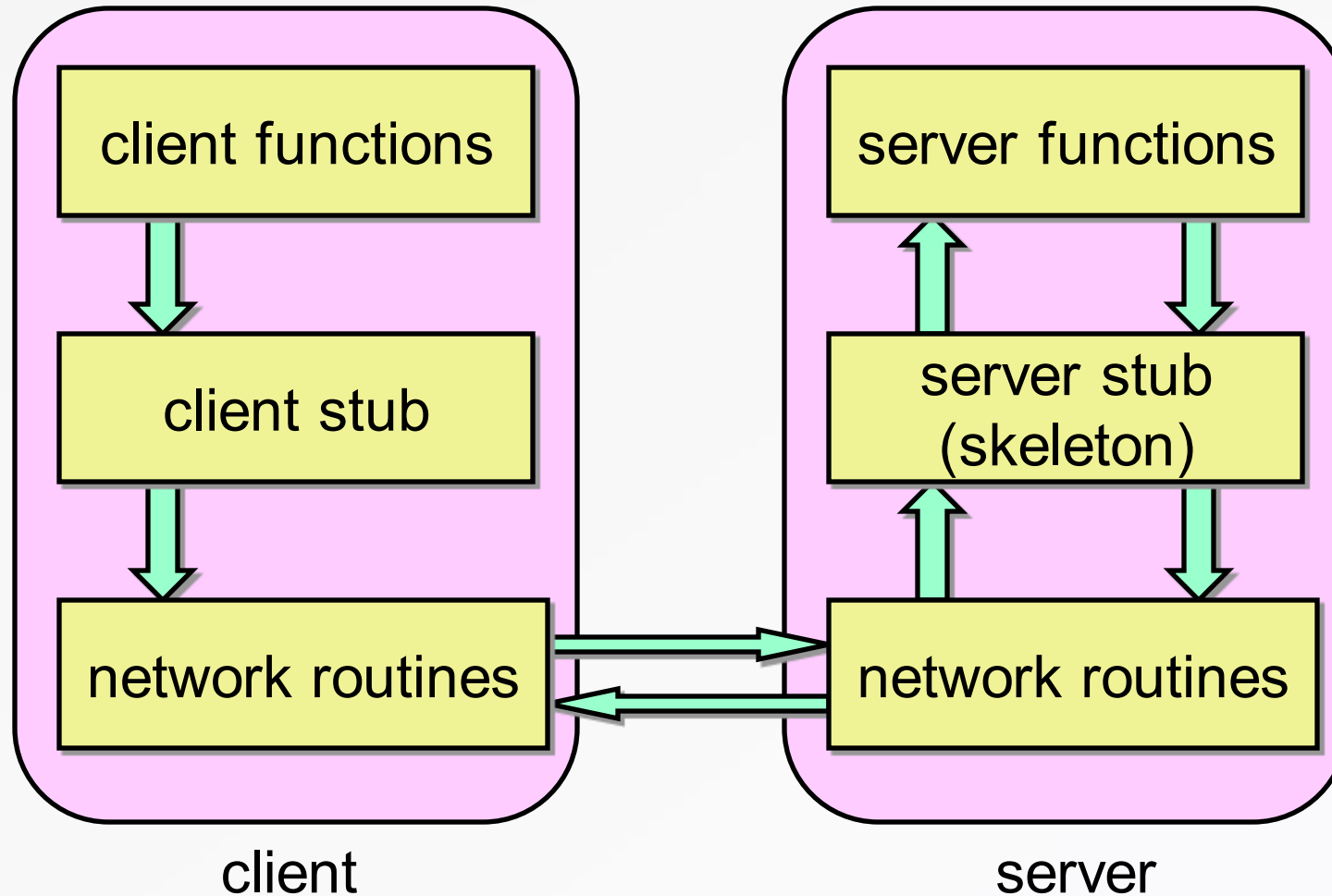
# Stub functions

## 7. Marshal return value and send message



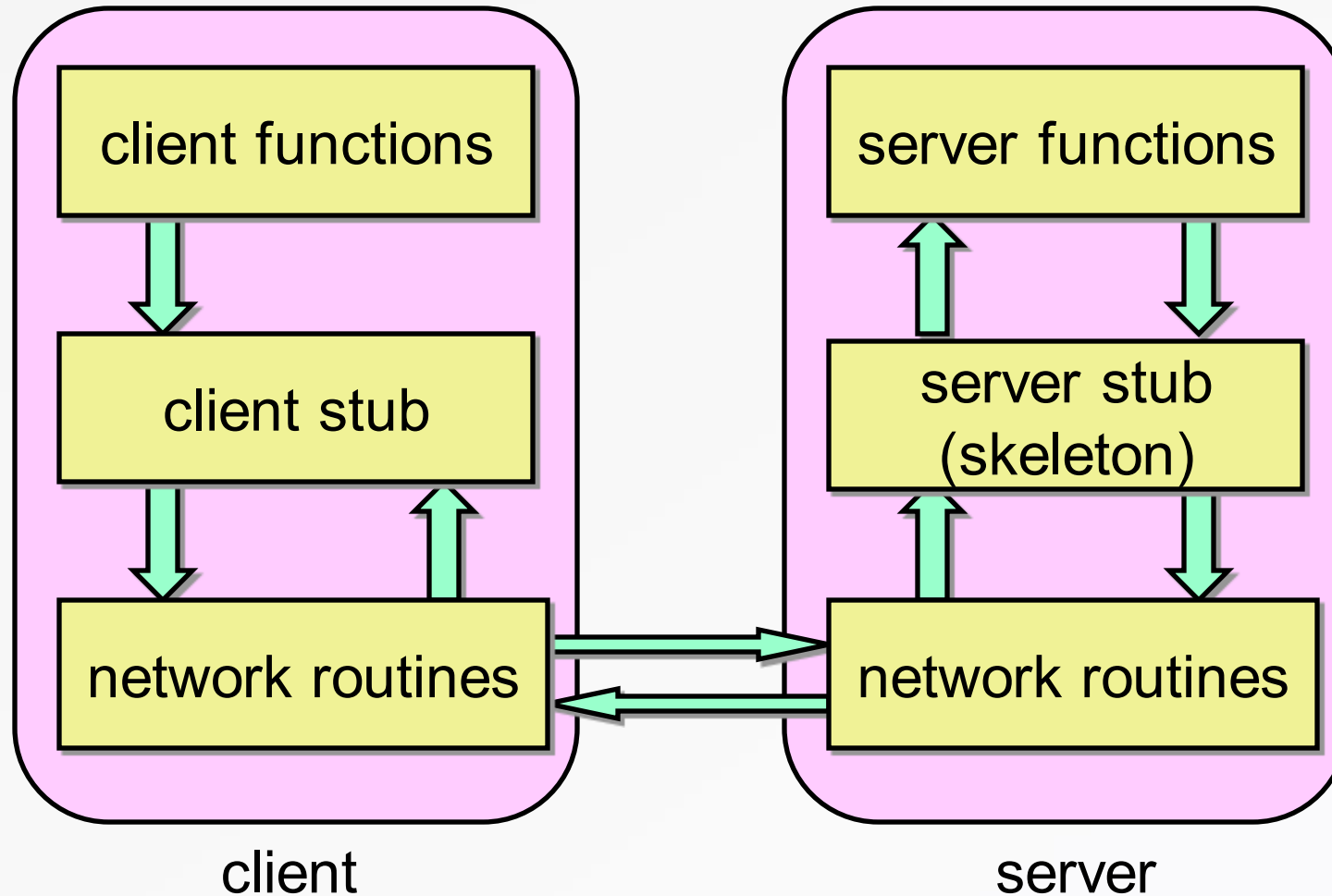
# Stub functions

## 8. Transfer message over network



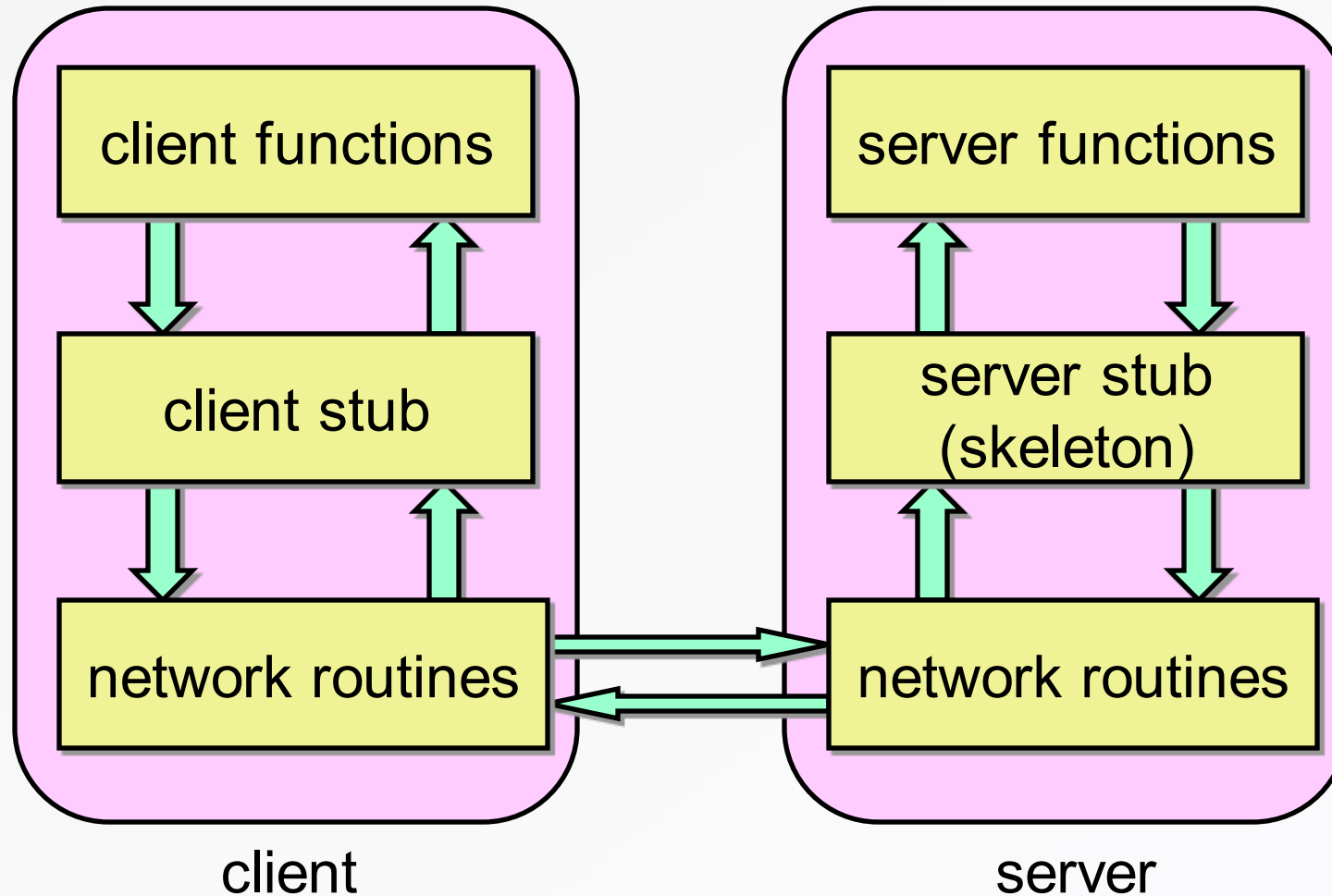
# Stub functions

## 9. Receive message: direct to stub



# Stub functions

## 10. Unmarshal return, return to client code





# Distributed Communication - Remote Procedure Calls Part 2

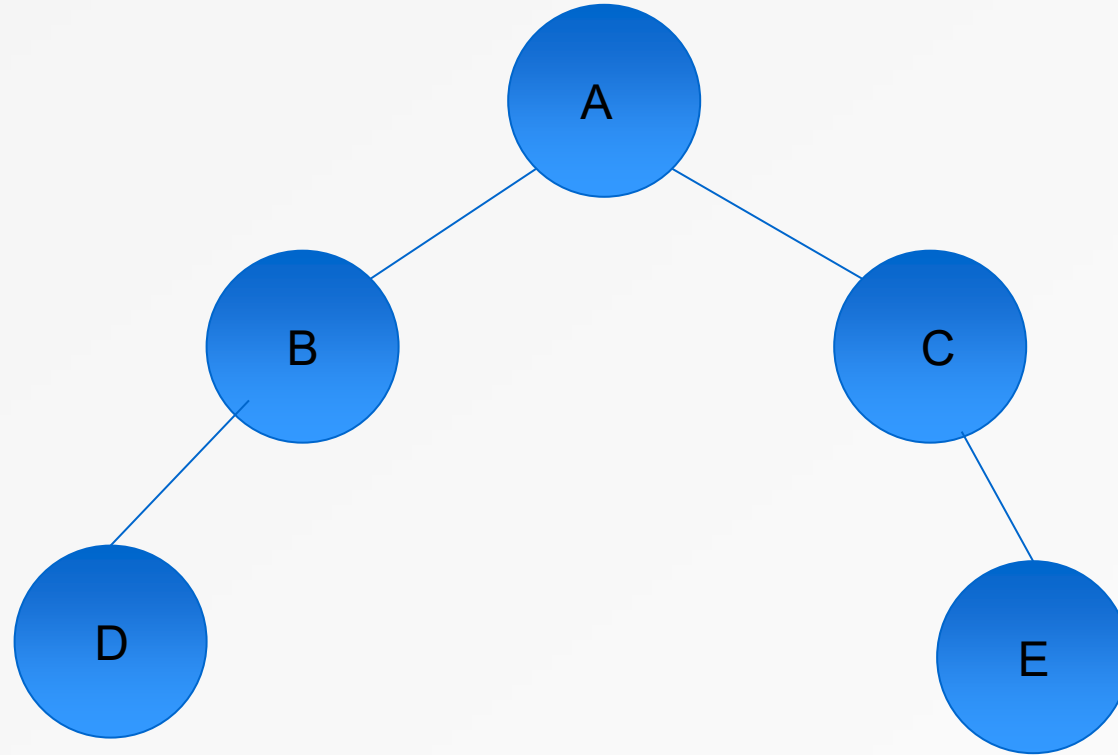
# Agenda

- Limitations
- Design Issues
- Call Semantics
- Implementation

# Limitations

- Parameters passed by values only and pointer values are not allowed.
- To support complex structures that use references
  - Copy structure into pointerless representation
  - Transmit
  - Reconstruct structure with local pointers on server

# Passing references-Trees and Graphs



- Tree: (A ( B ( D,- ), ( C ( - , E) ) -String version of a tree and traversal
- Graphs- Build adjacency matrix and convert into a string form.



# Representing Data

- Remote machine may have:
  - Different byte ordering
  - Different sizes of integers and other types
  - Different floating point representations
  - Different character sets
  - Alignment requirements

## Solution:

- Need standard encoding to enable communication between heterogeneous systems
- eXternal Data Representation (XDR) used by SUN RPC.

# Representing data

## Big endian vs Little endian byte ordering for 16 and 32 bit values

- Most significant byte in low memory
  - Sparc, 680x0, MIPS, PowerPC G5 (Big endian)
  - x86/Pentiums (Little endian)

```
main() {  
    unsigned int n;  
    char *a = (char *)&n;  
  
    n = 0x11223344;  
    printf("%02x, %02x, %02x, %02x\n",  
           a[0], a[1], a[2], a[3]);  
}
```

Output on a Pentium:  
44, 33, 22, 11

Output on a PowerPC:  
11, 22, 33, 44

# Failures

- Failure: RPC is vulnerable to failures.
- RPC involves communication network, another host and the process on that host.
- The programmer must know about various call semantics.
- The programs must handle errors that are likely to occur.

# Design Issues

## ➤ Delivery guarantees

- Retry request message: Do we need to retransmit the request message until we come to know either a reply or the server has failed.
- Duplicate filtering : when we retransmit the requests, whether we do filtering out duplicates at the server.
- Retransmission of reply messages: whether to keep a state (reply messages) to let lost reply messages to be retransmitted without re-executing the server operations.

# Call Semantics

## ➤ At-most-once semantics

- Either the remote procedure executes or it does not execute.
- Timer from client
- Either a server has crashed or request message from client has not reached.
- The server filters the duplicate requests that are caused by retransmissions due to network (channel) failure, slow or crashed server.
- The server also implements the reply cache

# Call Semantics Cont..

## ➤ 0-or-more call semantics.

- Either the procedure is not executed or executed more than once.
- Stateful servers
- This semantic has a problem with respect to Non-Idempotent functions
- Idempotent functions are those that yield the same value when executed multiple times, e.g. Squareroot function.
- Non-Idempotent function are those that yield different values when executed multiple times, e.g. withdrawal of money from an account or deposit of money to the account.

# Call Semantics Cont..

- Exactly-once semantics
  - The procedure has been called exactly once.
  - Semantics on a single machine.

# RPC Implementation

- Interface processing: integrate the RPC mechanism with client and server programs in some programming languages.
- Communication: sending and receiving request and reply messages.
- Binding: finding an appropriate server for a particular service.



# Interface Processing

- Language support
  - Most programming languages (C, C++, Java, ...) do not a concept of remote procedure calls
  - Language compilers will not generate client and server stubs
- What is the solution?
  - Use a separate compiler to generate stubs (pre-compiler)

# Interface Processing

- Marshalling and unmarshalling of arguments;
- Dispatching of request messages to the appropriate procedure in the server;
- Interface Definition Language compiler processes interface definitions written in an interface definition language (msg.x);
- Generate a client stub procedure (msg\_clnt.c);
- Generate a server stub procedure (msg\_svc.c);

# Interface Processing

- Use the signatures of the procedures in the interface to generate marshalling and unmarshalling operations (msg\_xdr.c).
- Generate procedure headings for each procedure in the service from the interface definition (msg.h).

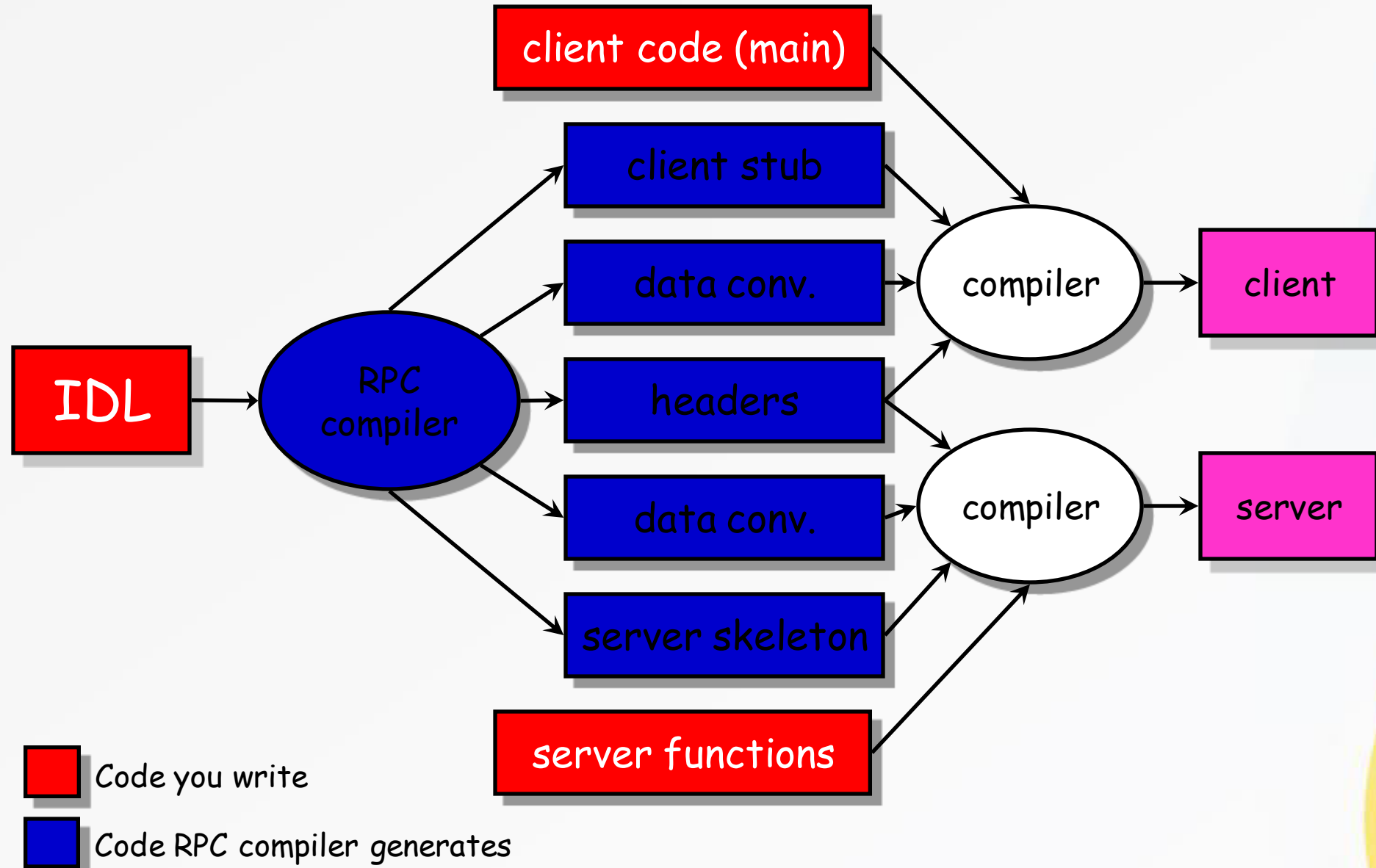
# Communication Handling

- TCP/UDP as transport protocols
- Socket Programming

# Binding

- An interface definition specifies a textual service name for use by clients and servers.
- Clients that request message must be addressed to a server port.
- Binder: a separate service that maintains a table containing mappings from service names to server ports.
- Binder interface used by server
  - Register (String serviceName, Port serverPort, int version)
  - Withdraw (String serviceName, Port serverPort, int version)
- Binder interface used by client
  - PortLookUp (String serviceName, int version)


# RPC compiler



# Clock Synchronization

The background of the slide features a gradient from dark blue at the top to light blue at the bottom. Overlaid on this are several wavy, horizontal bands of color. A prominent yellow band curves across the lower half of the image, with a lighter blue band just below it and a white band at the very bottom. The overall effect is a modern, clean, and abstract design.

# Agenda

- Physical Clocks
  - Need for Clock Synchronization
  - Physical Clock Synchronization
    - Internal Synchronization
    - External Synchronization
- 



# Time

- Time is one of the important issues in DS
  - Ex. At what time of day a particular event occurred at a particular computer, i.e., Event ordering
- There are certain problems in the real world that require algorithms to be developed for clock synchronization.
- Database updates, network timeouts, etc.
- Because of absence of a global clock, the notion of physical time is problematic in DS.

# Physical Clocks

- Let  $C(t)$  be a perfect clock.
- A clock  $C_i(t)$  is called correct at time  $t$  if  $C_i(t) = C(t)$ .
- A clock  $C_i(t)$  is called accurate at time  $t$  if
$$dC_i(t)/dt = dC(t)/dt = 1$$

# Physical Clocks Cont..

- Computers have physical clocks based on crystal oscillators.
- Clock skew
  - The time difference between two clocks.
- Clock drift: Crystal-based clocks count time at different rates, and therefore a variation in time (The difference in rates of two clocks).

# Physical Clocks Cont..

- The frequency with which these crystals oscillate differ because of the material with which these crystals are designed.
- Temperature could be another factor for the variation in clock's freq.
- If we observe the the difference in the oscillations between two clocks, it might be small, but the difference accumulated over many oscillations may lead to a notable difference.
- For clocks based on a quartz crystal, the drift is about  $10^{-6}$  sec/sec – giving a difference of one second every 1,000,000 sec or 11.6 days.

# Why Synchronization?

- Train reservation system
- Server A receives a client request to purchase last ticket on a train XYZ123.
- Server A completes the purchase using local clock with a timestamp 10h:25m:30s, and logs it. It sends a reply “ok” message to the client.
- That was the last seat. Server A sends message to Server B saying “train full.”
- Server B updated “Train XYZ123 full” + local clock value (which reads 10h:10m:10s) into its log.
- Server C queries A’s and B’s logs. C is confused that a client purchased a ticket after the train became full.
  - Incorrect or unfair actions.

# Processes and Events

- An Asynchronous Distributed System (DS) has a number of *processes*.
- Each process has a *state* (values of variables)
- Each process takes *actions* to change its state, which may be an *instruction* or a communication action (*send*, *receive*).
- An *event* is the occurrence of an action.
- Each process has a local clock – events *within* a process can be assigned *timestamps*, and thus ordered linearly.
- But – in a DS, we also need to know the time order of events across different processes.
- Clocks across processes are not synchronized in an asynchronous DS
  - Process clocks can be different
  - Need algorithms for either (a) time synchronization, or
  - (b) for telling which event happened before which.

# Physical Clocks Cont..

- Coordinated Universal Time (UTC).
- UTC is the “correct” time at any point of time.
- Assume that the UTC time is  $t$  and the process  $i$ 's time is  $C_i(t)$  then ideally one would like to have  $C_i(t) = t$ , or  $dC/dt = 1$ .
- In practice, a tolerance variable  $\rho$  is used such that,

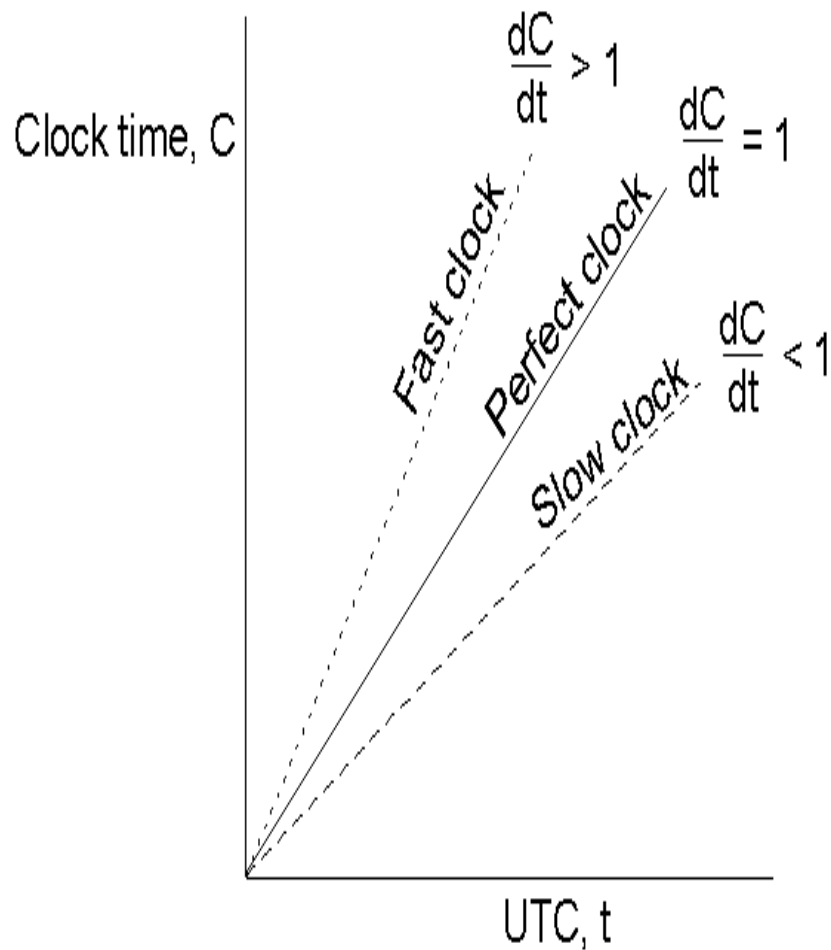
$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

# Types of Physical Clock Synchronization

- In external synchronization, clock is synchronized with an authoritative external source of time
- In internal synchronization clocks are synchronized with one another with a known degree of accuracy



# Physical Clocks Cont..



The relation between clock time and UTC when clocks tick at different rates.

- When a clock has a skew.
- Ex. It is 5 seconds behind.
  - Advance it 5 seconds to correct.
- Ex. It is 5 seconds ahead.
  - Pushing back 5 seconds is a bad idea.
  - Message was received before it was sent.
- We want monotonicity:
  - time always increases.

$\rho$  is specified by the manufacturer and is known as the maximum drift rate.

# Synchronization in Asynchronous system

- The Berkeley algorithm
- The Cristian's algorithm

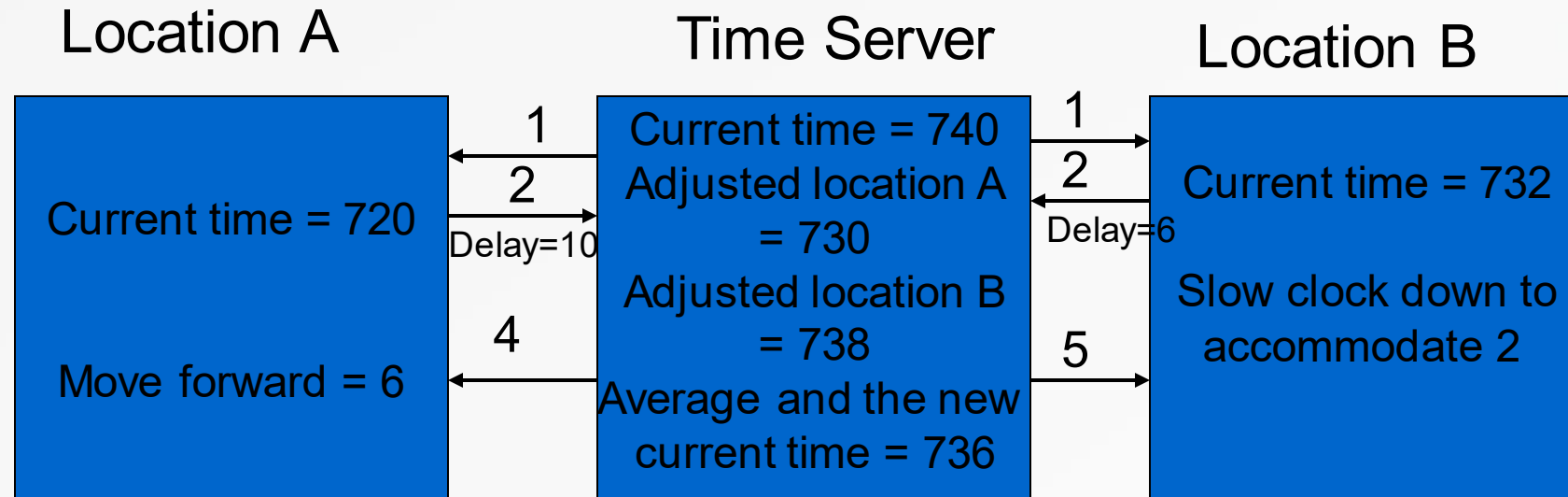
# The Berkeley algorithm

- Used for internal synchronization
- A coordinator (time server): *master*
- Periodically the *master* polls the time of each client (slave) whose clocks are to be synchronized.
- Based on the answer, it computes the average (including its own clock value) and broadcasts the new time.
- This method is suitable for a system in which no machine has a WWV receiver getting the UTC.

# The Berkeley algorithm

- The average cancels out the individual clock's tendencies to run fast or slow.
- The accuracy depends upon a nominal maximum Round Trip Time (RTT) between the master and the slaves.
- The master eliminates any occasional readings associated with larger times than this maximum.

# Example: Berkeley Algorithm

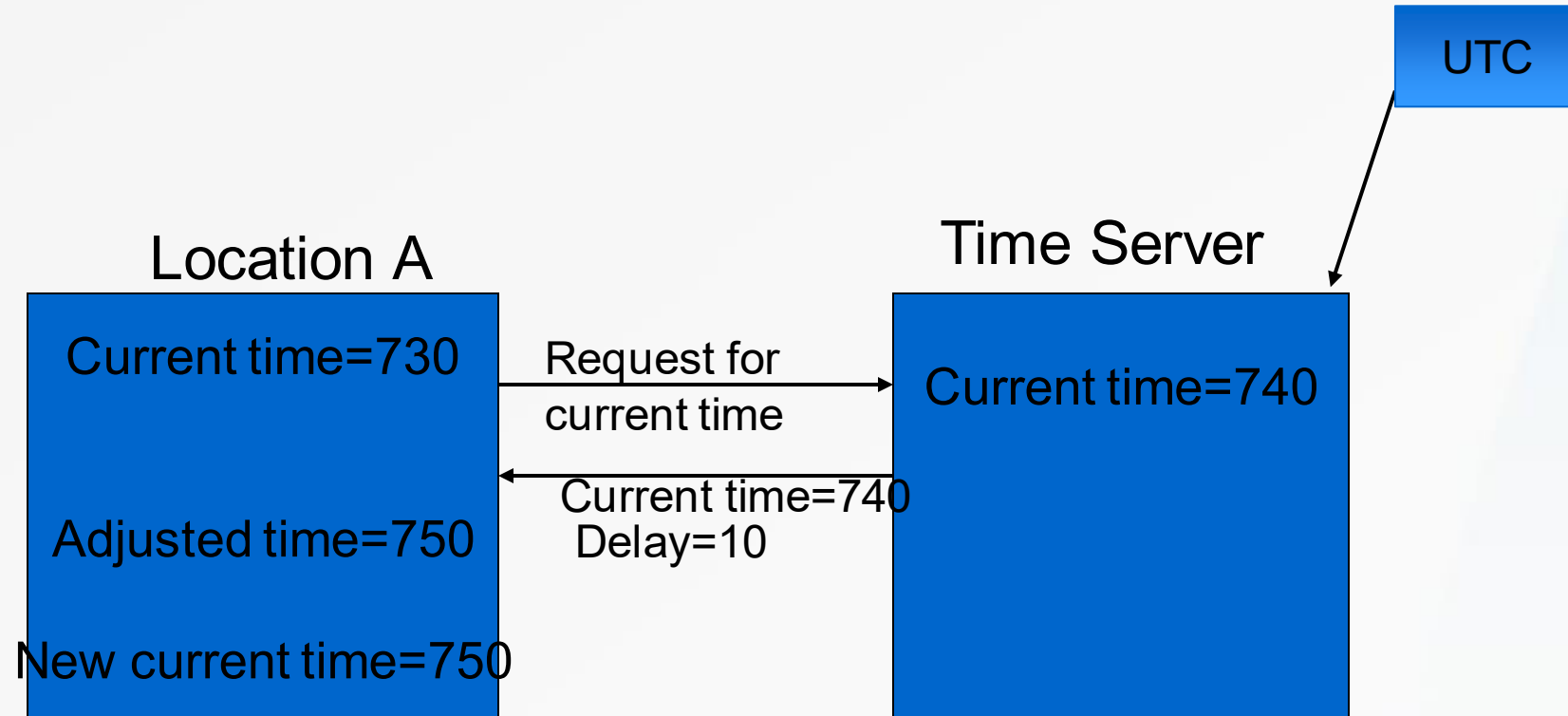


1. Current time = 740
2. My current time = 720
3. My current time = 732
4. Adjust forward = 6
5. Adjust slowdown to accommodate 2

# The Cristian's algorithm

- Used for external synchronization
- Periodically, each machine sends a request message *to a time server* and the server sends the time.
- The roundtrip message transmission delay time is computed, say this is  $T_{round}$ .
- The algorithm sets  $t + T_{round}/2$  as the new time.

# Example: The Cristian's algorithm



# References

George\_Coulouris\_Distributed\_Systems\_Concepts\_and\_Design\_5th\_Edition.



# Synchronization in Asynchronous system

Implement the file transfer between two nodes using UDP sockets

Transfer files block-by-block

Simulate the failure handling

# Logical Clocks

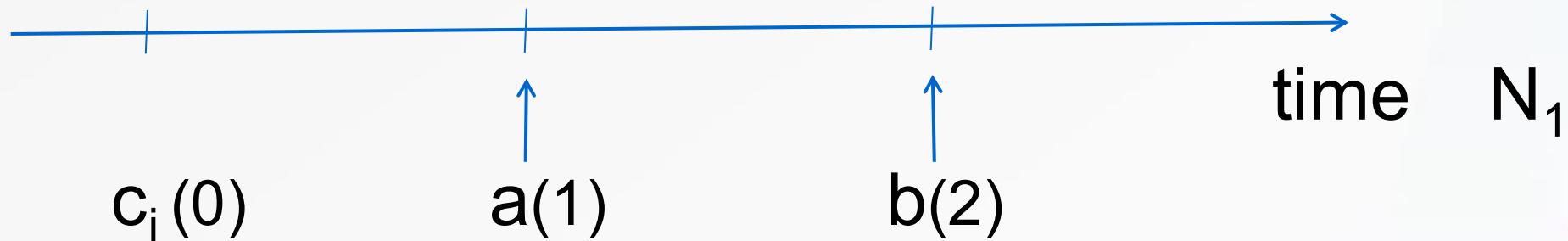


# Event Ordering

- Ordering of events is done only when there is some (relationship) causality among the events, i.e. one event causes the other event.
- Logical Clocks
  - Clocks that are logical and not the underlying physical clocks at nodes.
  - Event ordering can be done using logical clocks.
  - The “happened before” relationship.
  - Ex. Replicated Server Database: All updates must be applied to all copies of the server.

# Lamport's Logical Clock

- The non-negative integer  $C_i$  is maintained at every node  $N_i$ .
- Lamport's scalar clocks.
- Initially  $C_i$  is set to 0.



# Implementation Rules (IRs)

- If  $a$  and  $b$  are successive events at a node, then  $C_b^i = C_a^i + 1$
- If ' $a$ ' is an event of sending of a message  $m$  from node  $n_1$  and ' $b$ ' is an event of receipt of  $m$  at node  $n_2$ , then  $C_b^2 = \max C_b^2, t_m + 1$  where  $t_m$  is the timestamp of  $m$  and it is set to  $C_a^1$ .

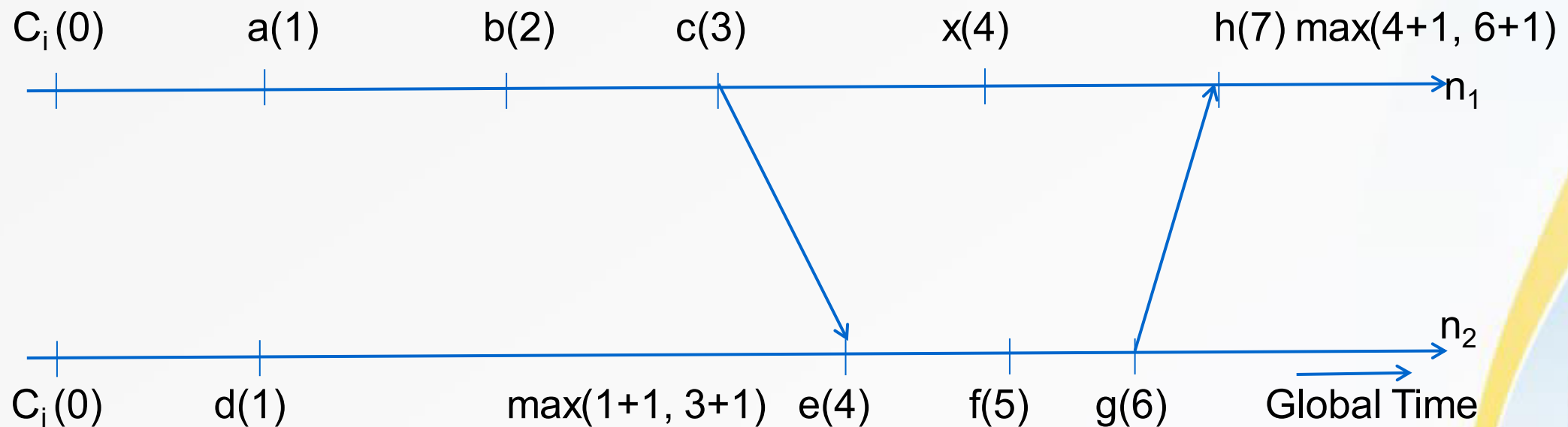
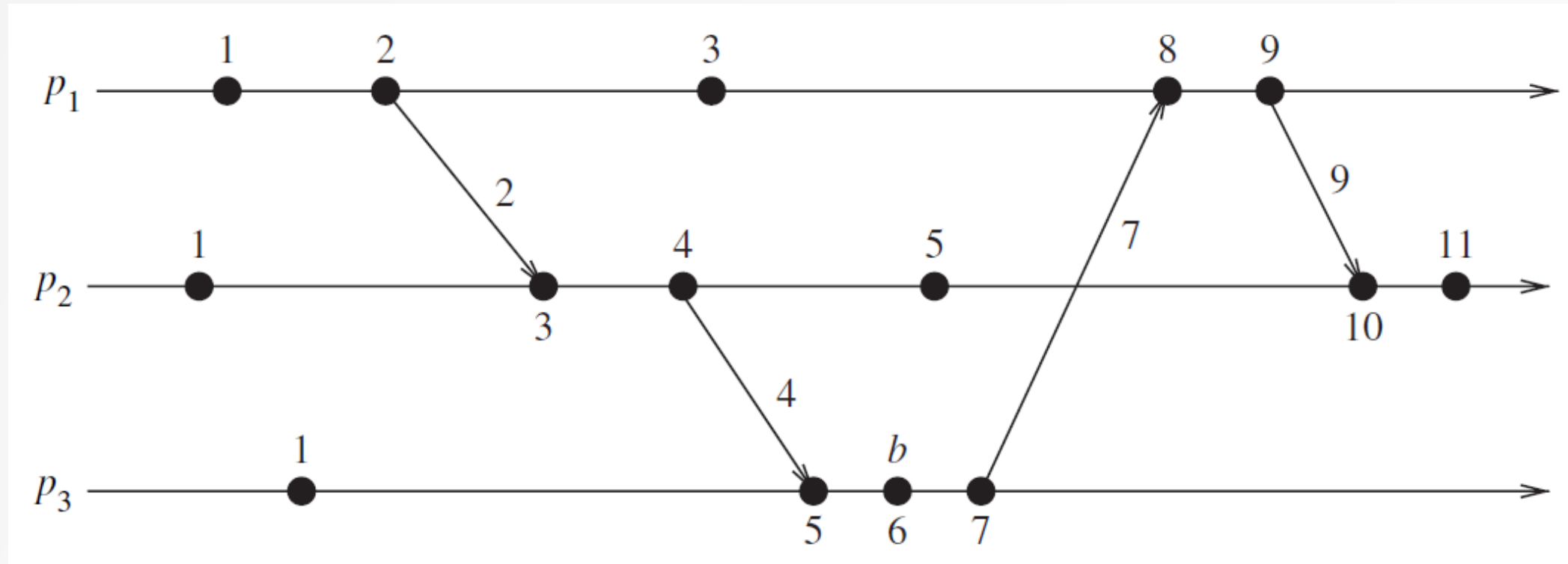


Fig. 1 Implementation of Lamport's Logical Clocks

# Example- Lamport's Logical Clocks



# Happened Before Relationship

1. If  $a$  and  $b$  are successive events at the same node, then we have  $a \rightarrow b$
2. If ' $a$ ' is the event of sending of a message  $m$  and ' $b$ ' is the event of receipt of  $m$ , then we have  $a \rightarrow b$
3. If  $a \rightarrow b$  and  $b \rightarrow c$  then we have  $a \rightarrow c$  (transitivity).  
Lamport's clock ensures that If  $a \rightarrow b$  then  $C_a < C_b$ .
4. If  $a \not\rightarrow b$  and  $b \not\rightarrow a$ , then  $a \parallel b$ , in this case  $a$  and  $b$  are called as concurrent events.

## Happened Before Relationship Cont..

- It imposes the partial order, i.e. the partial ordering of events is done.
- A few events have same timestamps, i.e., we don't know which happened before which event.
- E.g. a in  $N_1$  and d in  $N_2$  as shown in Fig. 1.



# Happened Before Relationship Cont..

## ➤ Total Ordering

- Lamport's logical clocks can be totally ordered.
- It can be done by assigning a tuple to every event
- Tuple :  $\langle c, d \rangle$  where  $c$  is a clock value and  $d$  is a node id.
- If the clocks have same values, then we differentiate using their node ids.
- Note: In the tuple, lower value indicates the higher priority.
- The total order relation  $<$  on two events  $x$  and  $y$  with timestamps  $(h, i)$  and  $(k, j)$ , respectively, is defined as follows:

$$x < y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

# Lamport's Logical Clocks

## ➤ Advantages:

- Lamport's logical clocks have a minimum overhead.
- Actions are performed only when the events happen.
- Happened before relationship tells us the potential causality.

## ➤ Limitations:

- If  $C_a < C_b$ , then  $a \rightarrow b$  is not true.
- If 'a' is the event at node  $N_1$  with timestamp 1 and 'b' is the event at node  $N_2$  with timestamp 2, then we can not say if 'a' has happened before 'b' even if  $C_a < C_b$ .

# Vector Clocks

- To overcome the limitations of Lamport's Logical Clocks and for keeping track of transitive dependencies among processes, vector clocks were proposed.
- A node  $N_i$  maintains a vector  $V_i[1...n]$ , where there are  $n$  nodes in the system.
- Rules for updating vector clocks.
  1. If  $a$  and  $b$  are successive events at  $N_i$  then

$$V_b^i[i] = V_a^i[i] + 1$$

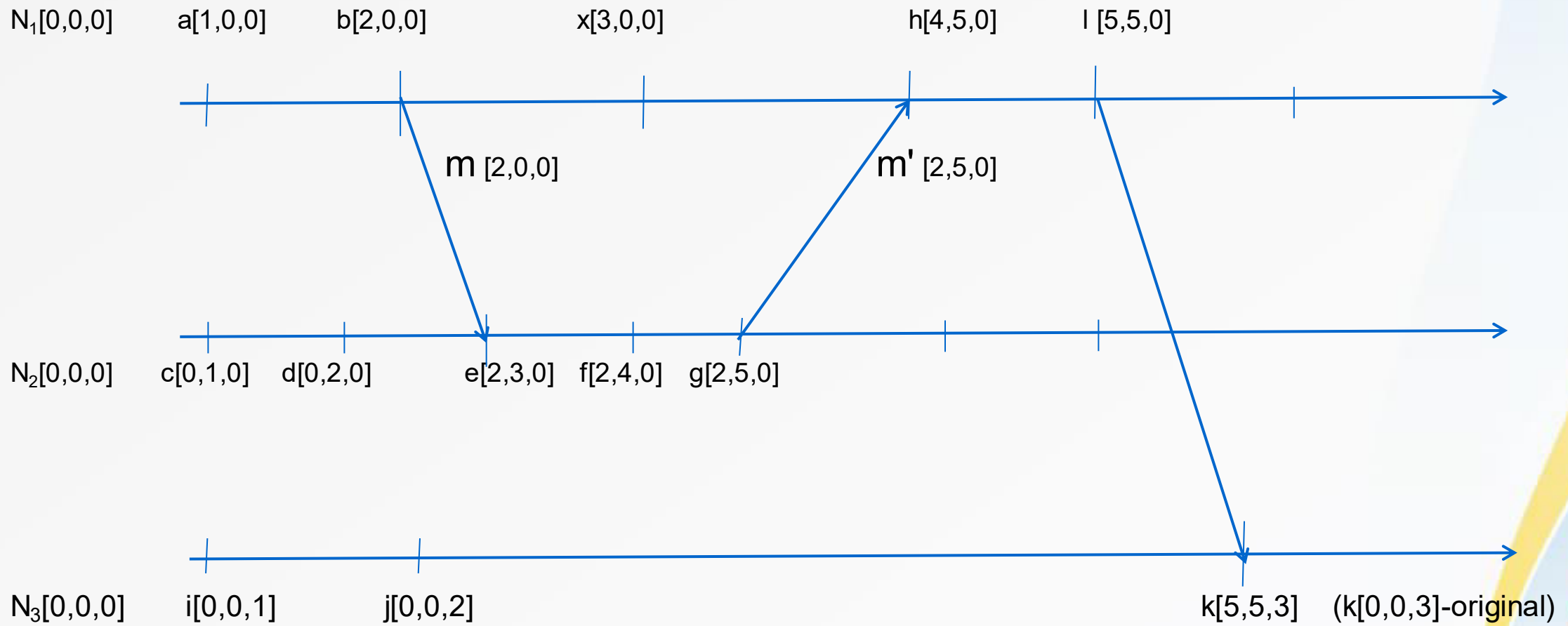
- $i^{\text{th}}$  entry of event  $a$  in vector is updated.

## Vector Clocks Cont..

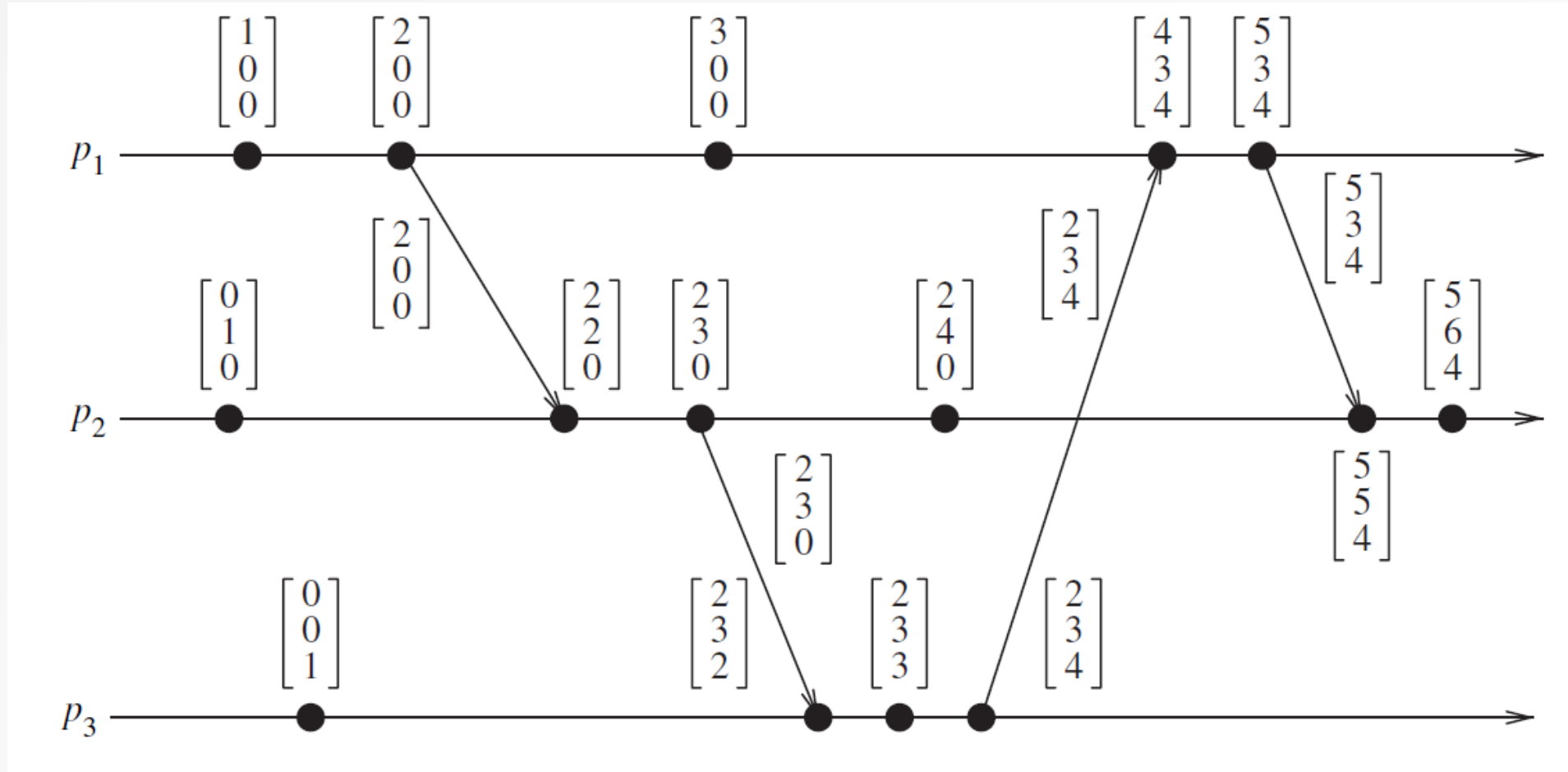
2. If 'a' is an event of sending of a message  $m$  from node  $N_i$  and 'b' is an event of receipt of  $m$  at node  $N_j$ , then

- (i).  $v_a^i[i]$  is set according to (1).
- (ii).  $m$  is assigned a timestamp  $t_m = v_a^i[i]$
- (iii).  $v_b^j$  is set according to (1).
- (iv).  $v_b^j[x] = \max(v_b^j[x], t_m[x])$

# Example-Vector Clocks



# Example-Vector Clocks



# Vector Comparison Rules

- (i).  $v_1 = v_2$ , if  $\forall_i 1, 2, \dots, n \ v_1[i] = v_2[i]$ .
- (ii).  $v_1 \neq v_2$ ,  $\exists_i \forall_i 1, 2, \dots, n \ v_1[i] \neq v_2[i]$ .
- (iii).  $v_1 \leq v_2$ , if  $\forall_i 1, 2, \dots, n \ v_1[i] \leq v_2[i]$ .
- (iv).  $v_1 < v_2$ , if  $v_1 \leq v_2$  and  $v_1 \neq v_2$

Example:

➤ If  $i[0 \ 0 \ 1]$  and  $d[0 \ 2 \ 0]$ , then  $v_i \neq v_d$

In this case,  $i \parallel d$ , i.e. both are concurrent.

➤ if  $b[2 \ 0 \ 0]$  and  $g[2 \ 5 \ 0]$ , then  $v_b < v_g$ , hence  $b \rightarrow g$ .

➤ IFF  $x \rightarrow y$  then  $V_x < V_y$  (Converse is true as well).

# Vector Clocks

- Limitations
- Additional overhead: Size of a message and vector are flowing in the message (Space).
- How many nodes in the system?
- Ex. WhatsApp : Any number of nodes can be added or deleted (256 is the maximum size of a group).



# References

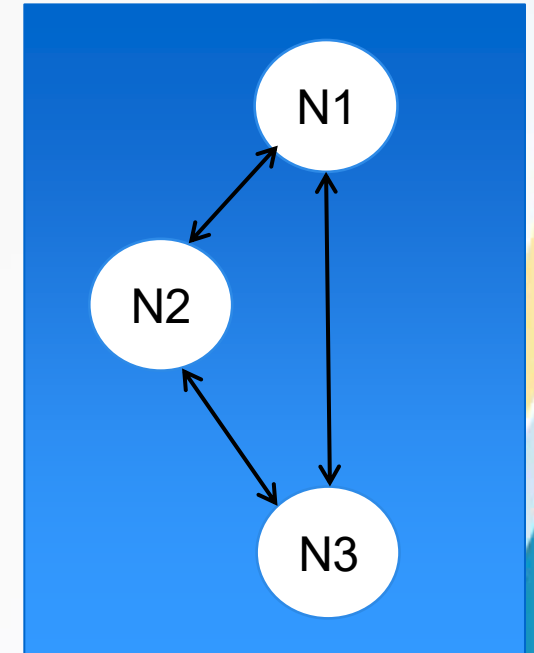
- Advanced Concepts In Operating Systems by Mukesh Singhal and Niranjana Shivratni (Tata McGraw-Hill Education).
- Distributed Computing Principles, Algorithms, and Systems by Ajay D. Kshemkalyani and Mukesh Singhal

# Applications of Vector Clocks

The background of the slide features a gradient from dark blue at the top to light blue at the bottom. Overlaid on this are several wavy, horizontal bands of color. A prominent yellow band curves across the middle, with a lighter blue band underneath it. The bottom of the slide transitions into a white area.

# Causally Ordered Broadcast

- Causally ordered broadcast is one of the applications of vector clocks.
- $N_1$ ,  $N_2$  and  $N_3$  are mirror replicas of servers.
- The order in which updates happen.
- The causal ordering of messages was first proposed by Birman and Joseph and was implemented in ISIS (Commercial DS).



# Causally Ordered Broadcast Cont..

- Assume N3 sends a broadcast msg  $m_1$  to  $N_1$  &  $N_2$ .
- Assume N2 sends a broadcast msg  $m_2$  to  $N_1$  &  $N_3$ .
- We also have  $m_1 \rightarrow m_2$ ,  $\text{Send}(m_1) \rightarrow \text{Send}(m_2)$ , i.e.,  $m_1$  has happened before  $m_2$ .
- $m_1$  must be given to application at a server replica first.
- $m_2$  must be given to application at a server replica next.
- If  $m_2$  is received before  $m_1$  is received, then  $m_2$  must be buffered and given to application after  $m_1$  is applied or given to an application.
- How to ensure these message ordering?

# Birman-Schiper-Stephenson (BSS) Protocol

➤ Broadcast-based protocol.

➤ Protocol

1. When a node  $N_i$  has a message  $m$  for broadcast, it performs the following actions.

(i). Increment  $V_i[i]$

Timestamp  $m$  with  $t_m = V_i$ .

2. When a node  $N_j$  receives  $\langle m, t_m \rangle$  from  $N_i$ , it performs the following actions.

(i). If  $(V_j[i] < t_m[i] - 1)$  OR

$V_j[k] < t_m[k]$  (for any  $k$  between 1 and  $n$  except  $i$ )

Buffer  $m$

# Birman-Schiper-Stephenson Protocol Cont..

else

    deliver  $m$  to the application

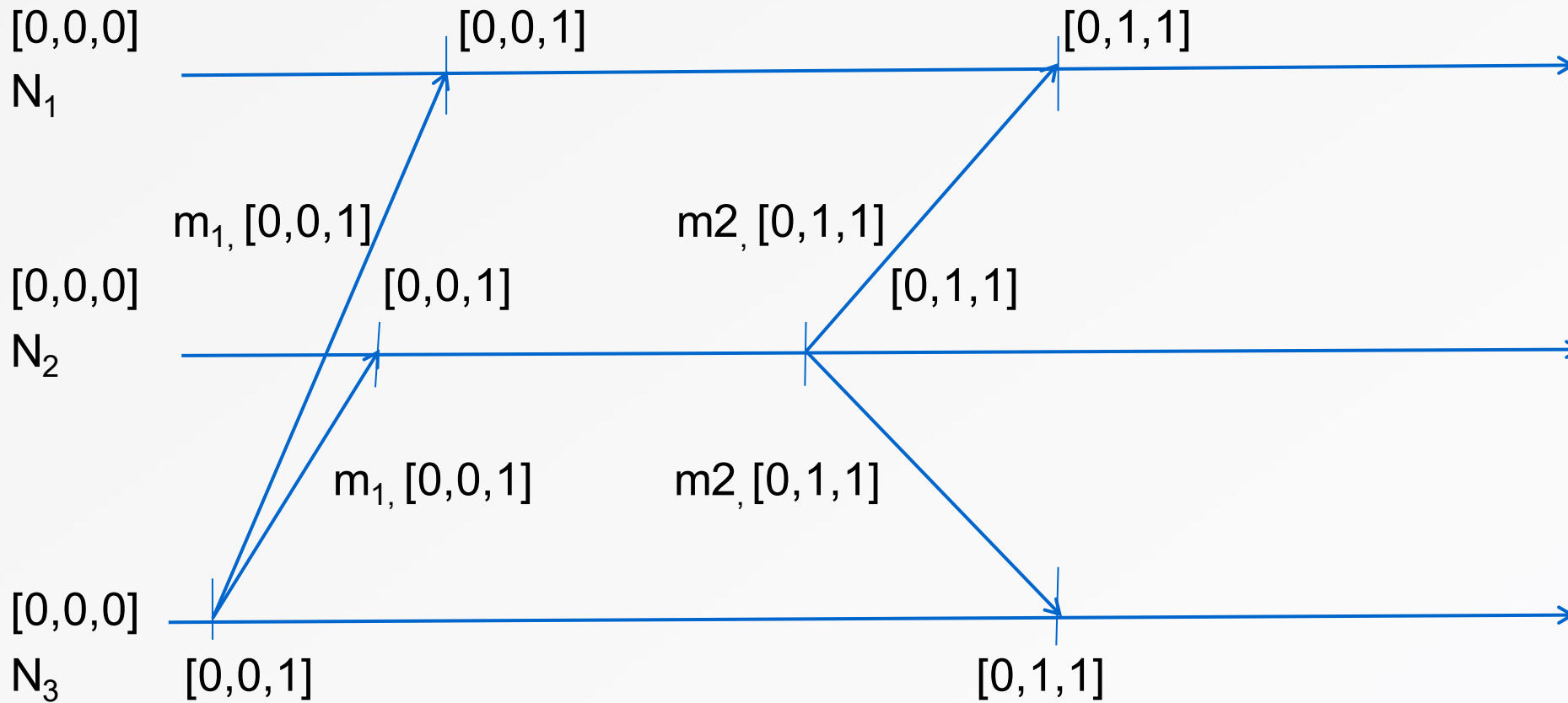
Messages are buffered in the order of their timestamps.

Concurrent messages are buffered in the order of their receipt.

(ii). If  $m$  is delivered, then  $V_j[i]$  is incremented.

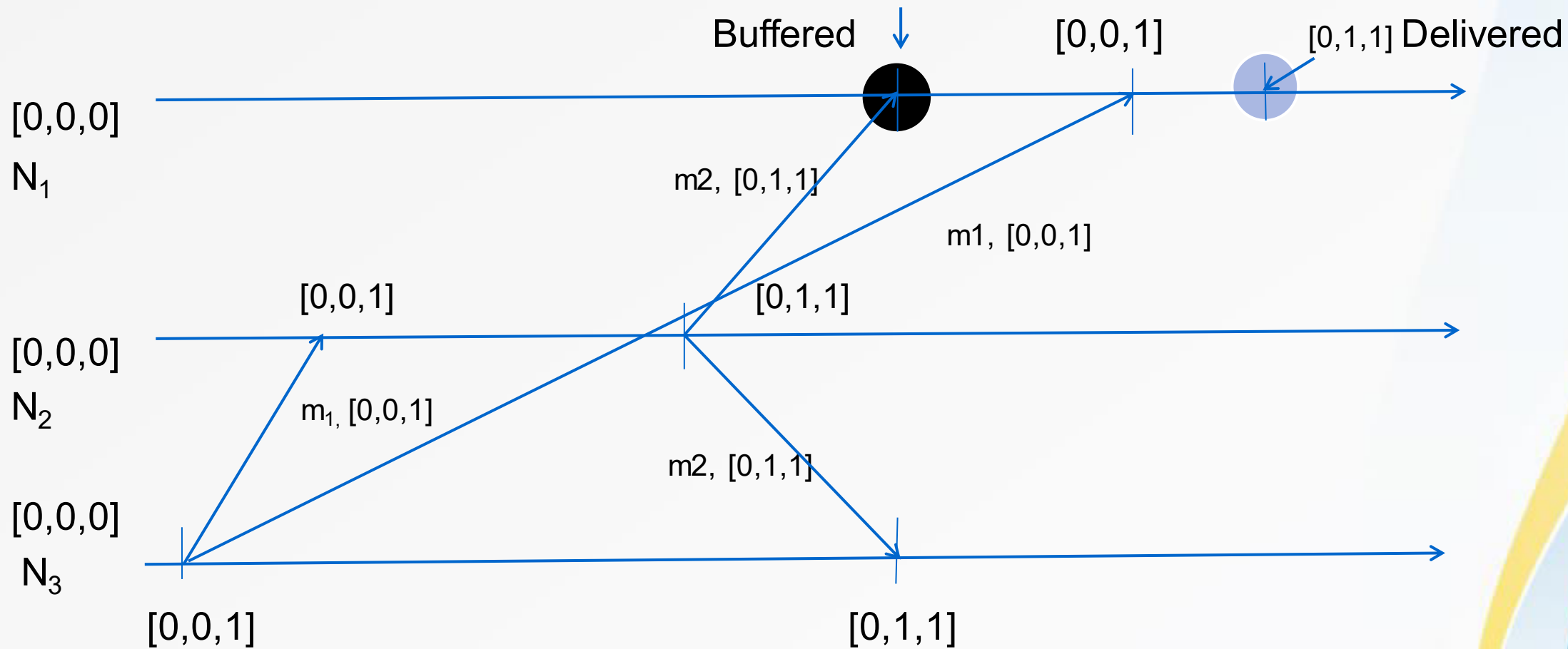
    Check now if any buffered message can be delivered.

# Example- BSS Protocol



$V_j[i] < t_m[i] - 1$  and  $V_j[k] < t_m[k]$  both are false for  $m_1$  and  $m_2$ , so messages will not be buffered, they are given to applications.

# Example- BSS Protocol



$V_j[i] < t_m[i] - 1$  is false, but  $V_j[k] < t_m[k]$  is true for  $m_2$ , so  $m_2$  will be buffered and not given to application until  $m_1$  is received by  $N_1$  and given to an application.



# Birman-Schiper-Stephenson Protocol Cont..

## ➤ Step 2 (i).

➤ Part a ensures that the process  $P_j$  on node  $N_j$  has received all the messages from  $P_i$  (on node  $N_i$ ) that precede  $m$ .

➤ Part b ensures that  $P_j$  has received all those messages received by  $P_i$  before sending  $m$ .

## ➤ Concurrent Messages:

➤  $V_1 \parallel V_2$  iff  $V_1 \not\prec V_2$  and  $V_2 \not\prec V_1$

# Applications of Vector Clocks



# Scipher-Eggli-Sandoz (SES) Protocol

- Causally ordered Unicast (More General Case).
- Using the unicast to causally order the messages.
- Protocol basics
  - A node  $N_i$  maintains a vector timestamp  $V_i$  and another vector  $S_i$
  - Entries in  $S_i$  are in the form  $\langle N_j, t \rangle$  which means that  $N_i$  sent a message to  $N_j$  with vector timestamp  $t$ .
  - Entries in  $S_i$  are stored in the order of their timestamps.

# Scipher-Eggli-Sandoz Protocol Cont..

- Protocol

(1). When a node  $N_i$  has a message  $m$  to be sent to  $N_j$ , it performs the following actions.

a). Increment  $V_i[i]$ .

Set  $t_m = V[i]$ , Set  $V_m = S_i$

b). Insert  $\langle N_j, t_m \rangle$  in  $S_i$

(An earlier entry for  $N_j$  in  $S_i$  is overwritten).

c). Send  $\langle m, t_m, V_m \rangle$  to  $N_j$

## Scipher-Eggli-Sandoz Protocol Cont..

(2). When a node  $N_j$  receives a message  $\langle m, t_m, v_m \rangle$  from  $N_i$ , it performs the following actions.

(a). If  $V_m$  does not contain any entry for  $N_j$ ,  
 $m$  is delivered.

Else if the entry is  $\langle N_j, t \rangle$  then

if  $t \geq V_j$

$m$  is buffered

else

$m$  is delivered.

# Scipher-Eggli-Sandoz Protocol Cont..

- b). If  $m$  is delivered, then merge  $S_j$  and  $V_m$  according to following rules.
- i). If  $S_j$  does not contain an entry for  $N_k$  ( $k \neq i$ ) and  $V_m$  contains an entry  $\langle N_k, t \rangle$  then  
Insert  $\langle N_k, t \rangle$  into  $S_j$ .  
Else Let the entry in  $S_j$  be  $\langle N_k, t' \rangle$ ,  
This entry is replaced with another entry  $\langle N_k, t_{\max} \rangle$ , where  $t_{\max}$  is computed as follows.  
$$\forall x, t_{\max}[x] = \max(t[x], t'[x])$$
- ii). Update  $V_j$  according to vector clock rule
- iii). Check if any buffered messages at  $N_j$  can now be delivered since the logical clocks are updated

# Example-SES Protocol Cont..

- Case-1:  $N_1$  [0 0 0] and  $m_1$  is a message to send.

1. a). Increment  $V_i[i] : V_i = [1 \ 0 \ 0]$

Set  $t_m = V[i] : t_m = 1$

Set  $V_m = S_i : V_m = \text{Null}$

b). Insert  $\langle N_j, t_m \rangle$  in  $S_i$

(An earlier entry for  $N_j$  in  $S_i$  is overwritten).

$S_i = \langle N_j, 1 \rangle$  // This is the first entry

c). Send  $\langle m, t_m, V_m \rangle$  to  $N_j : \langle m_1, 1, \text{Null} \rangle$  to  $N_j$

2. a. Since  $V_m$  does not contain an entry for  $N_j$ , the message can be safely delivered at  $N_j$  as this is the first message from  $N_i$  to  $N_j$

as

no causality problem here.

# Example-SES Protocol Cont..

- Case-2:  $N_1$  [1 0 0] and  $m_2$  is a message to send.

1. a). Increment  $V_i[i] : V_i = [2 \ 0 \ 0]$

Set  $t_m = V[i] : t_m = 2$

Set  $V_m = S_i : V_m = \langle N_j, 1 \rangle$

b). Insert  $\langle N_j, t_m \rangle$  in  $S_i$

(An earlier entry for  $N_j$  in  $S_i$  is overwritten).

$S_i = \langle N_j, 2 \rangle$  //  $\langle N_j, 1 \rangle$  is overwritten

c). Send  $\langle m, t_m, V_m \rangle$  to  $N_j : \langle m_2, 2, \langle N_j, 1 \rangle \rangle$  to  $N_j$

2. a. Else if the entry is  $\langle N_j, t \rangle$  then

if  $t \geq V_j$ , then buffer it

else deliver it.

( $1 \geq 2$  false, so deliver it)



# Example-SES Protocol Cont..

- Case-2:  $N_1$  [1 0 0] and  $m_2$  is a message to send.

1. a). Increment  $V_i[i] : V_i = [2 \ 0 \ 0]$

Set  $t_m = V[i] : t_m = 2$

Set  $V_m = S_i : V_m = \langle N_j, 1 \rangle$

b). Insert  $\langle N_j, t_m \rangle$  in  $S_i$

(An earlier entry for  $N_j$  in  $S_i$  is overwritten).

$S_i = \langle N_j, 2 \rangle$  //  $\langle N_j, 1 \rangle$  is overwritten

c). Send  $\langle m, t_m, V_m \rangle$  to  $N_j : \langle m_2, 2, \langle N_j, 1 \rangle \rangle$  to  $N_j$

2. a. Else if the entry is  $\langle N_j, t \rangle$  then

if  $t \geq V_j$  then buffer it

else deliver it

( $1 \geq 1$  true, so buffer it)

# Example-SES Protocol Cont..

- Case-2:  $N_1$  [1 0 0] and  $m_1$  is a message to send to  $N_3$
- $N_1$  [2 0 0] and  $m_2$  is a message to send to  $N_2$

1. a). Increment  $V_i[i]$  :  $V_i = [2\ 0\ 0]$

Set  $t_m = V[i]$  :  $t_m = 2$

Set  $V_m = S_i$  :  $V_m = \langle N_k, 1 \rangle$

b). Insert  $\langle N_j, t_m \rangle$  in  $S_i$

(An earlier entry for  $N_j$  in  $S_i$  is overwritten).

$S_i = \langle N_k, 1 \rangle$

c). Send  $\langle m, t_m, V_m \rangle$  to  $N_j$  :  $\langle m_2, 2, \langle N_k, 1 \rangle \rangle$  to  $N_j$

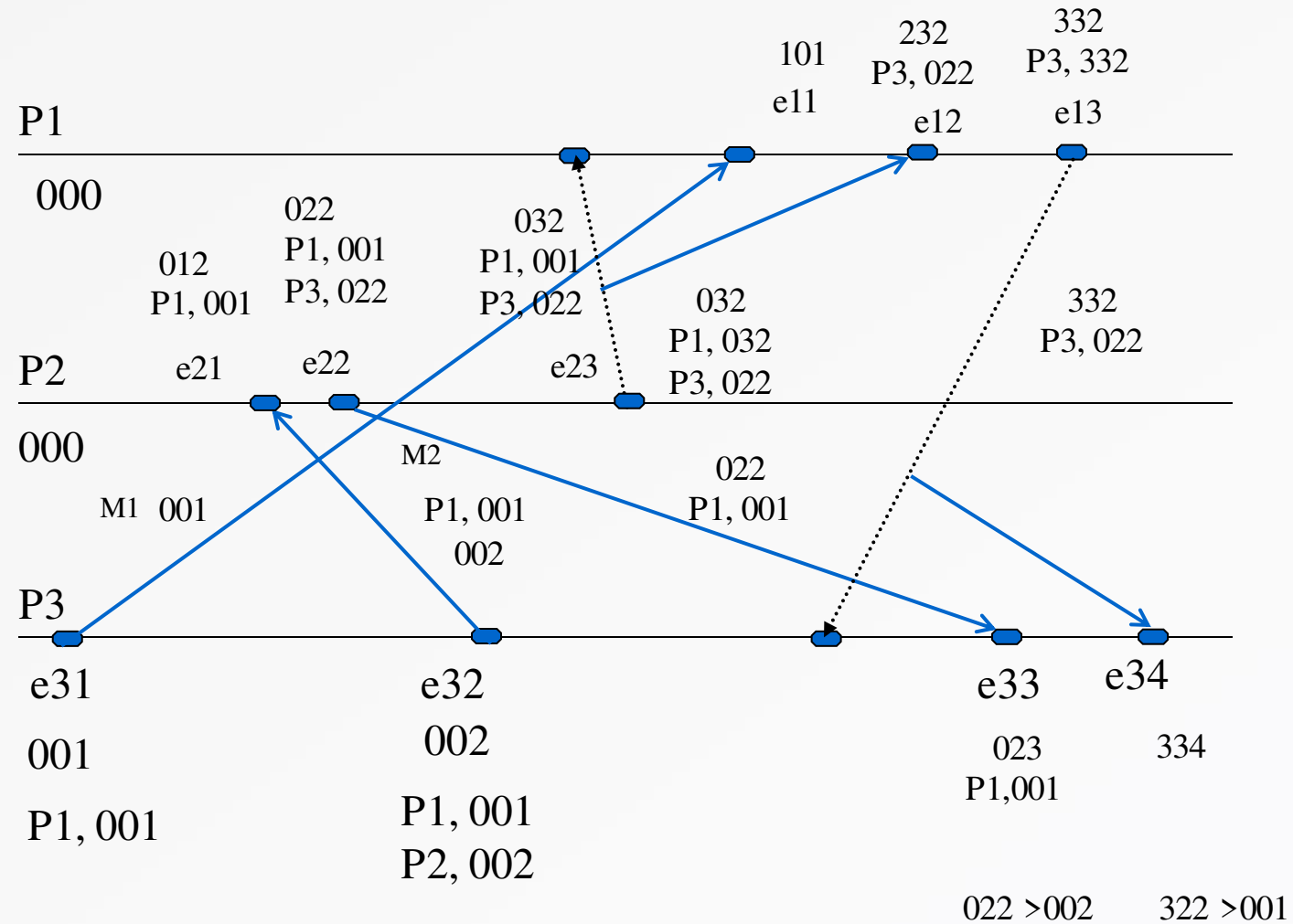
2. a.  $m_2$  can be delivered at  $N_j$

b.  $S_j$  does not contain an entry for  $N_k$  ( $k \neq i$ ) and  $V_m$  contains it  
insert  $\langle N_k, 1 \rangle$  in  $S_j$

# Example-SES Protocol Cont..

- Case-2:  $N_1$  [1 0 0] and  $m_1$  is a message to send to  $N_3$
  - $N_1$  [2 0 0] and  $m_2$  is a message to send to  $N_2$
1. a). Increment  $V_i[i] : V_i = [2 \ 0 \ 0]$   
Set  $t_m = V[i] : t_m = 2$   
Set  $V_m = S_i : V_m = \langle N_k, 1 \rangle$   
b). Insert  $\langle N_j, t_m \rangle$  in  $S_i$   
(An earlier entry for  $N_j$  in  $S_i$  is overwritten).  
 $S_i = \langle N_k, 1 \rangle$   
c). Send  $\langle m, t_m, V_m \rangle$  to  $N_j : \langle m_2, 2, \langle N_k, 1 \rangle \rangle$  to  $N_j$
  2. a.  $m_2$  can be delivered at  $N_j$   
b.  $S_j$  contains an entry for  $N_k$  ( $k \neq i$ ) and  $V_m$  contains it  
 $S_j = \langle N_k, 2 \rangle$  and  $V_m = \langle N_k, 1 \rangle$ , then  $t_{\max}[x] = \max(t[x], t'[x]) = 2$

# Example-SES Protocol



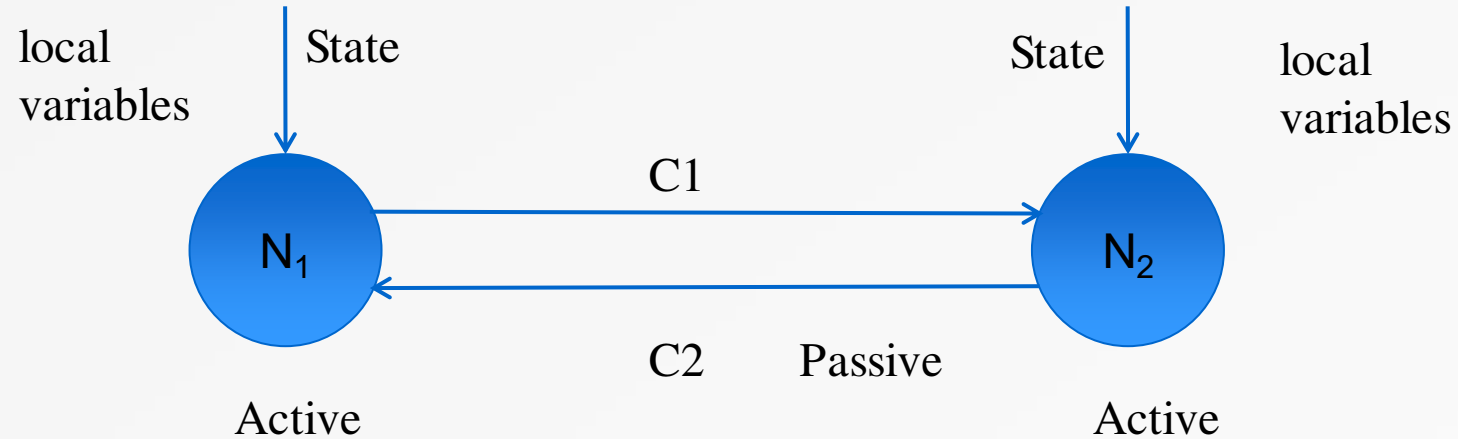
# Global System State Collection

The background features a gradient from dark blue at the top to light blue at the bottom. Overlaid on this are several wavy, horizontal bands of color: a light blue band, a yellow band, and a white band, creating a sense of movement and depth.

# State in Distributed System

- For a centralized system, the log is kept on the secondary storage.
- In case of a Distributed System, the state comprises of
  - Local state of each process and messages that are sent by a node together with the messages that are currently in transit, that is, that have been sent but not delivered.
- The processes in a distributed system in general would have only a limited view of the system, which consists of its local state and the messages it has sent or received.
- Difficult to take the state at each node at the same time because of absence of a global clock and imperfect clock synchronization.

# State in Distributed System



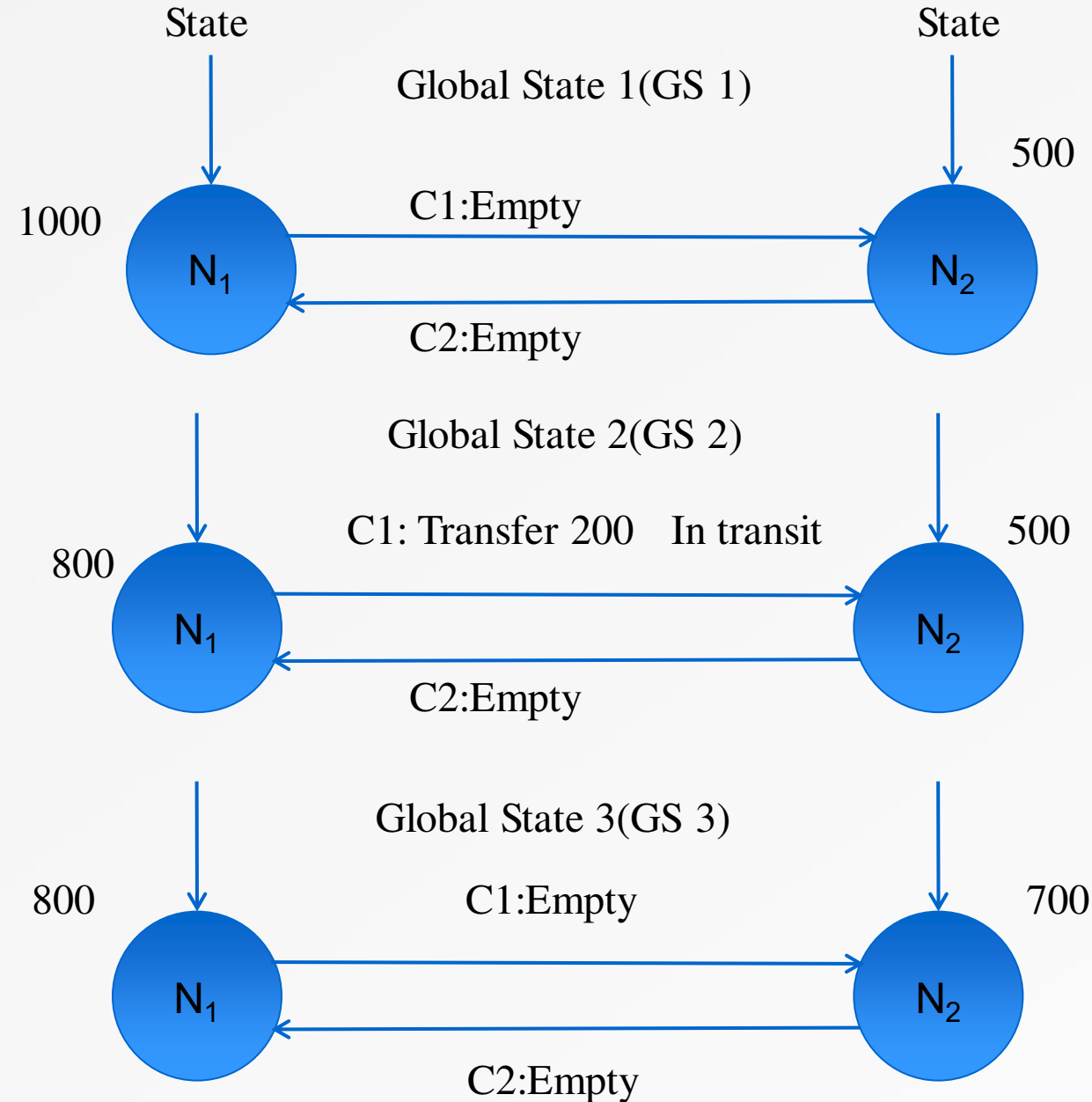
- States of  $N_1$  and  $N_2$  and the state of the channel itself.
- Some messages might be in transit.
- $N_1$  sends a message that is on the channel and has not been received by  $N_2$ .
- When to collect the state?
- Difficult to find the perfect global state as it requires the communication and computation

# Why do we need to record/collect state??

- Distributed Deadlock Detection.
- Distributed Debugging



# State in Distributed System Cont..



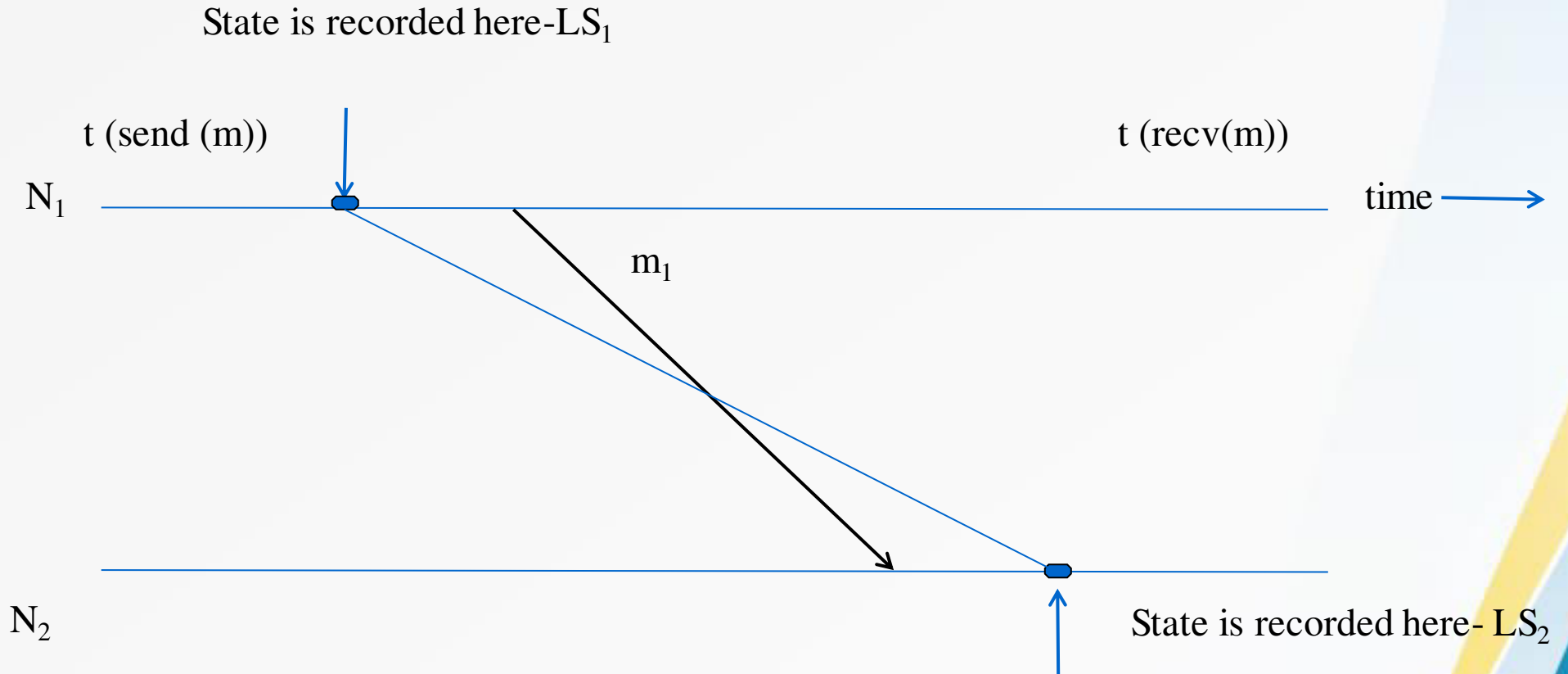
## ➤ Case 1

- State collected at  $N_1$  when the initial amount is Rs. 1000 (GS1)
- $N_1$  sends Rs. 200 to  $N_2$
- Channel state collected (GS2), message (Sending of Rs. 200) is on the channel, i.e. in transit
- State collected at  $N_2$  when the amount is Rs. 500 (GS1)
- Total amount in the system is 1700.

## ➤ Case 2

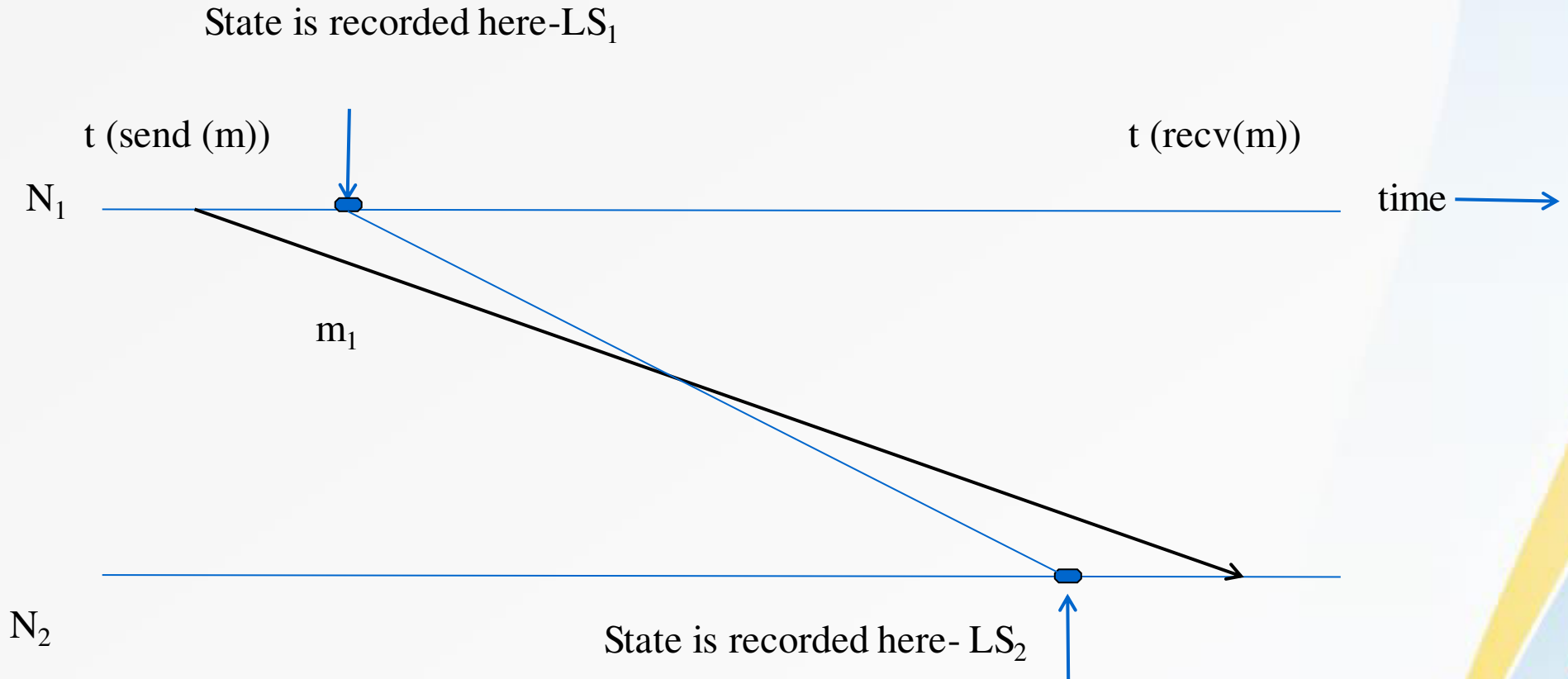
- Channel ( $C1$ : Empty) state collected at GS1.
- State of  $N_1$  (800) and  $N_2$  (500) collected at GS2.
- Total amount in the system is 1300.

# Distributed Cut



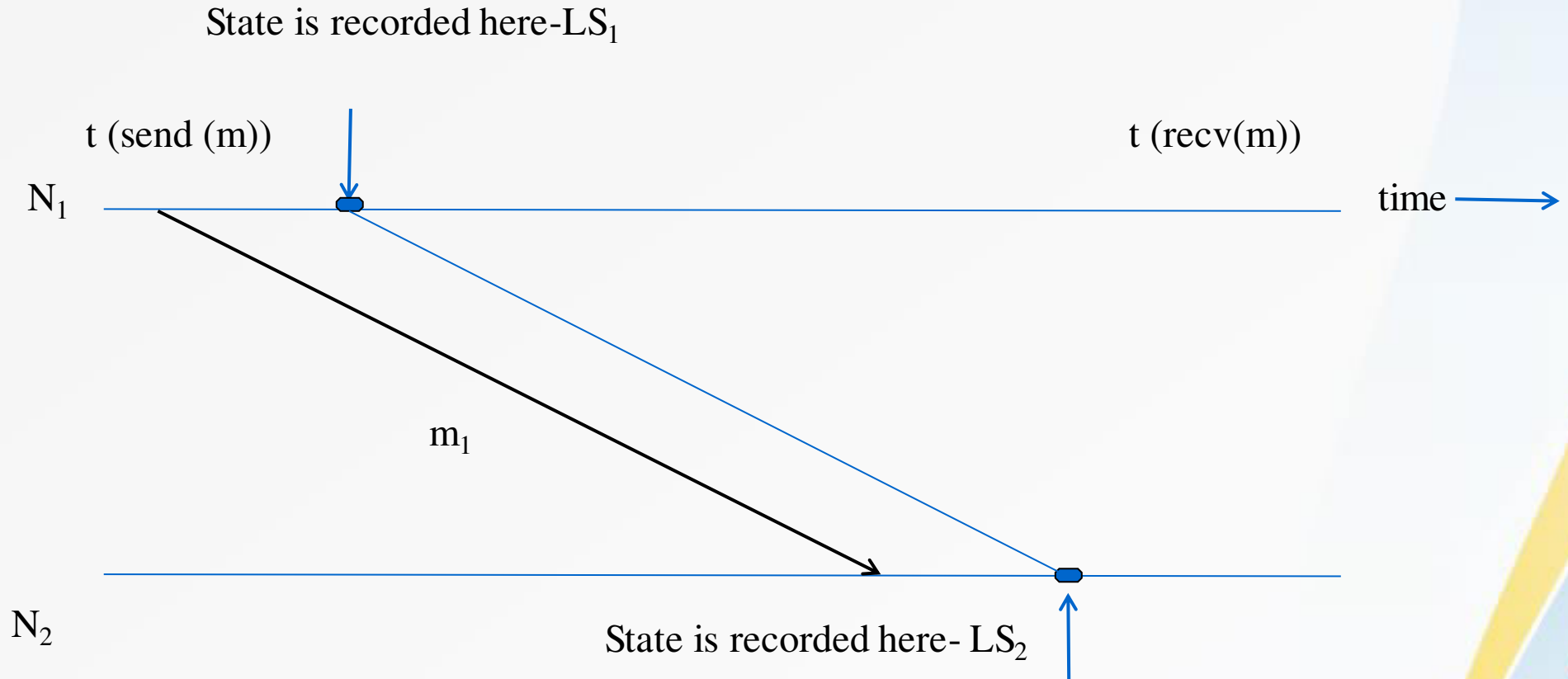
- $N_2$  receives the message and  $N_1$  says it did not send.
- If the distributed cut (G) has received the message, then that message is not consistent and the state is also not consistent.

# Distributed Cut Cont..



- $m_1$  is recorded in  $LS_1$ ,  $m_1$  is in transit.
- $m_1$  is not recorded in  $LS_2$
- The state is called as consistent with  $m_1$  in transit.

# Distributed Cut Cont..



- The perfect situation.
- The state is called as consistent and transitless.

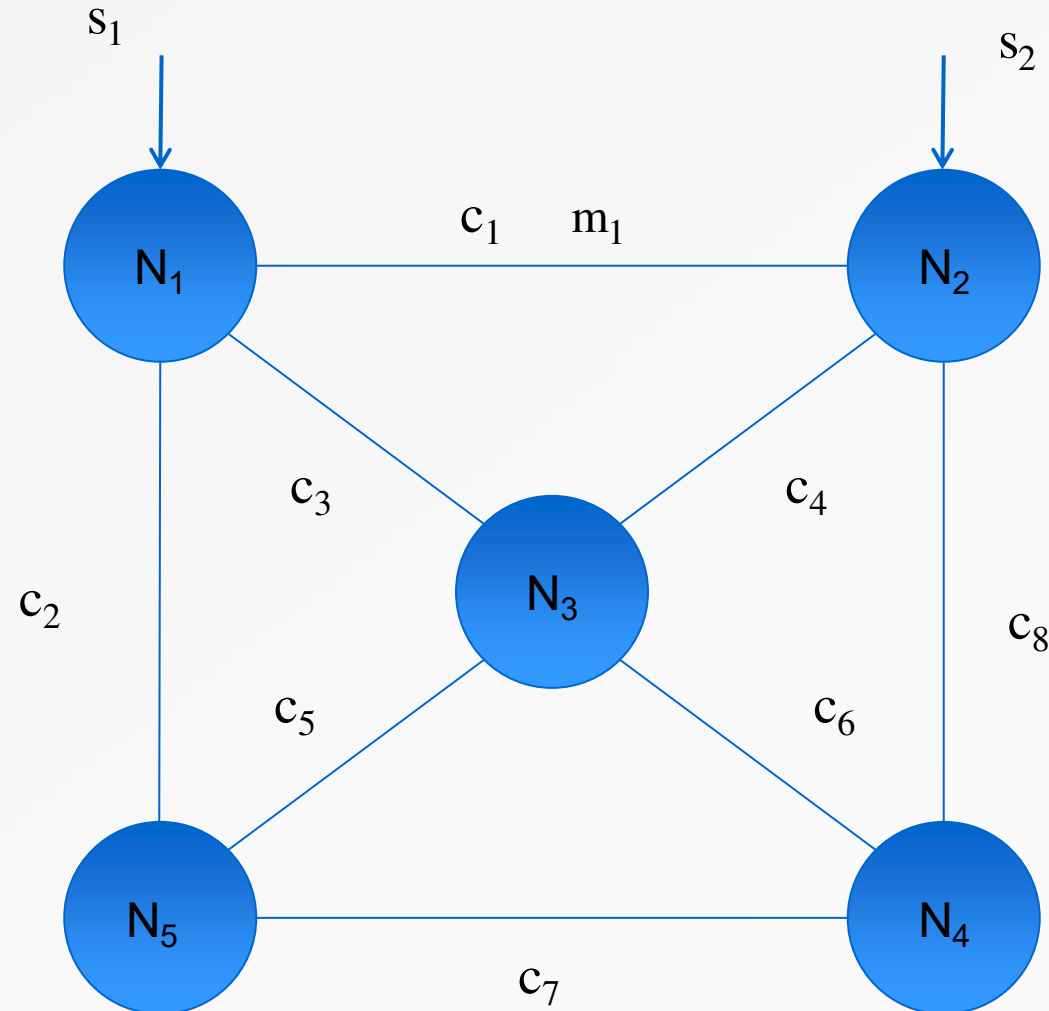
# Chandy-Lamport Global State Collection Algorithm

## ➤ Assumptions

- FIFO channels are required (TCP).
- Connected topology.
- Asynchronous system (No bound on message latency)
- No failures
  - No node failures and no message failures
- Record the states locally
- Any process can begin the global state recording algorithm

# Chandy-Lamport Global State Collection Algorithm

- M - Marker or Control Message
- M does not interfere with a data message.
- C - Channel



- Provides the consistent global state but does not guarantee actual state.

# Protocol

1. When a node  $N$  decides to initialize global state collection, it performs the marker sending rule mentioned below.

Marker sending rule for  $N$

- a).  $N$  records its state
- b). For each outgoing channel ( $C$ ) incident on  $N$ ,  $N$  sends marker msg before it sends any further computation msg.

# Protocol Cont..

2. When a node  $N'$  receives marker msg on channel  $C$ , it performs the following actions.
  - a). If  $N'$  has not recorded its state, then
    - i).  $N'$  records the state of  $C$  as empty.
    - ii).  $N'$  follows the marker sending rule.
  - b). Else
    - i). (Marker indicates that the channel state must be recorded)  
 $N'$  records state of  $C$  as all those computation msgs that it had received after it recorded its state and before it received the marker msg.
3.  $N'$  is finished when it has received a marker along each of its incoming channels.



## More details cont..

- When we say the algorithm is over?
- If a process  $N_j$  receives the marker requesting for recording the state for the first time, it considers the process that sent the marker as its predecessor.
- When  $N_j$  completes its part of the state recording, it sends its predecessor a DONE message.
- When the initiator receives a DONE message from all its successors, it knows that the state has been completely recorded.

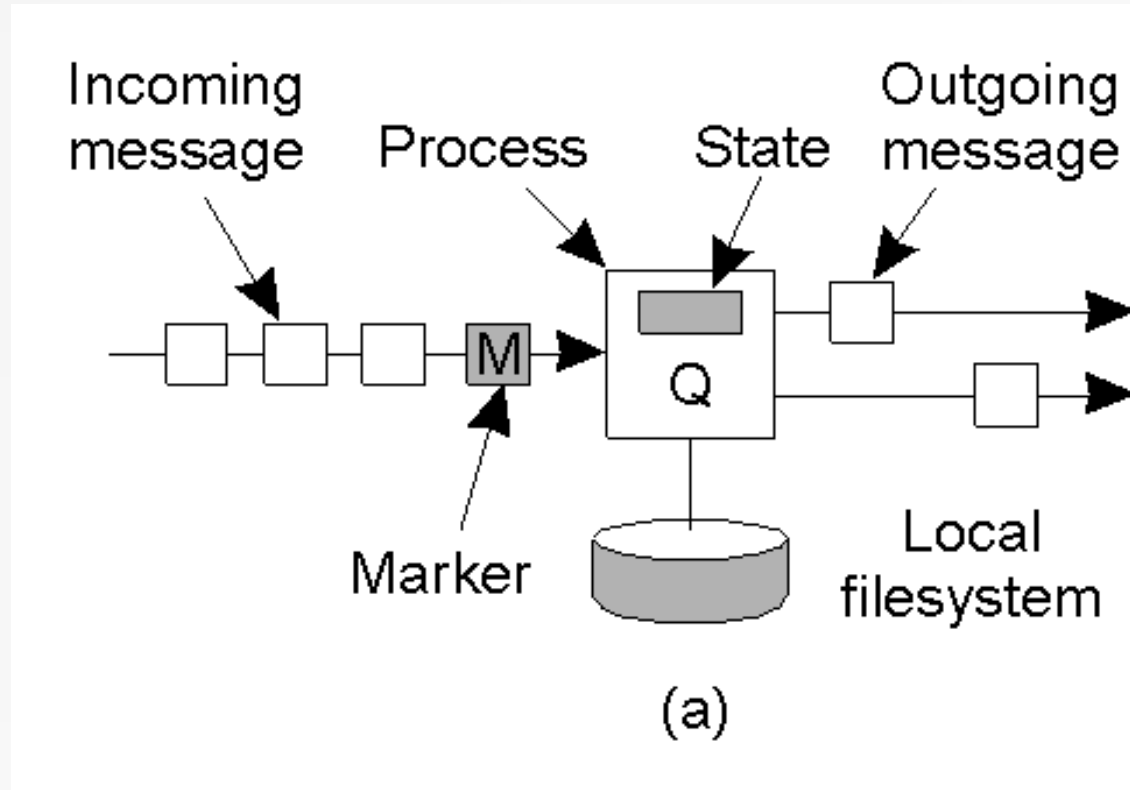
# More details

- The network is considered as a bidirectional graph.
- Marker msg goes from  $N_1$  to  $N_2$  (outgoing) and  $N_2$  to  $N_1$  (incoming)
- Contents of Marker message?
  - <Initiator id, Sequence Number>

## State collection activity

- The algorithm can be initiated by any node at any time and may have multiple initiations going concurrently.
- The consistent state that a node gets may be different for every node.

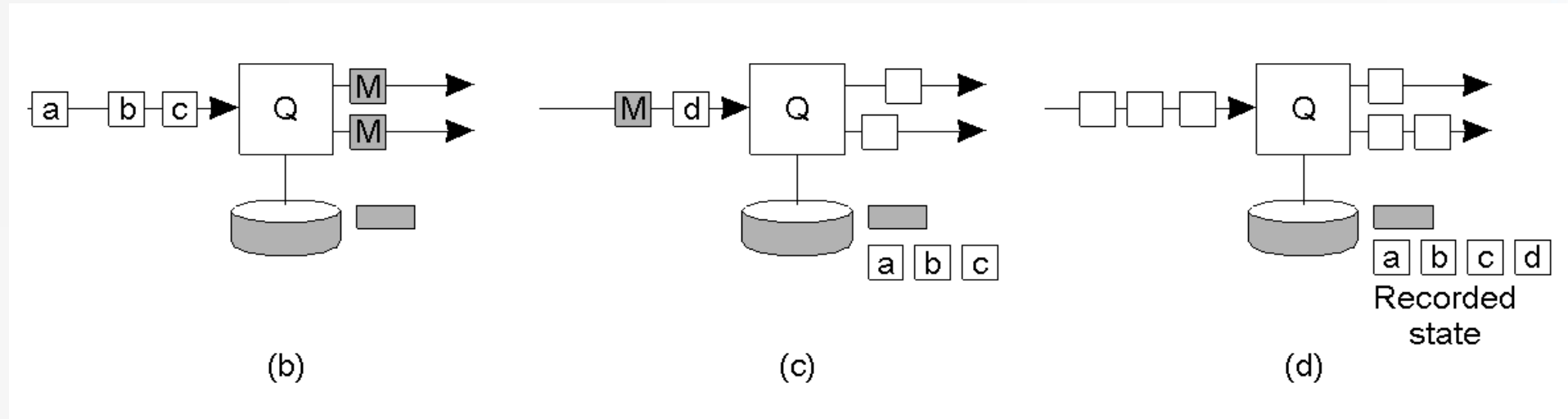
# Algorithm



- a) Organization of a process and channels for a distributed snapshot

Reference: Western Science- Computer Science (<https://www.csd.uwo.ca/>)

# Algorithm

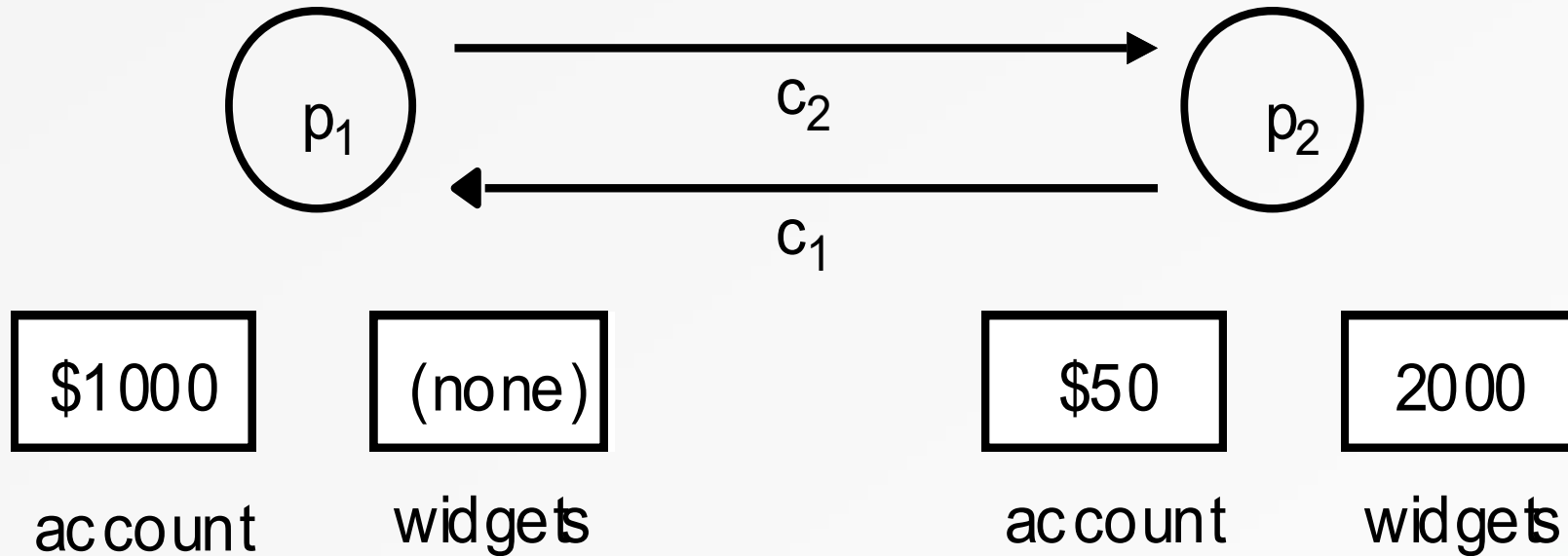


- b) Process Q receives a marker for the first time and records its local state
- c) Q records all incoming messages
- d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel

# Example

- Assume two process  $p_1$  and  $p_2$  connected by two unidirectional channels,  $c_1$  and  $c_2$ .
- The two processes trade in widgets.
- Process  $p_1$  sends orders for widgets over  $c_2$  to  $p_2$ , enclosing payment at the rate of \$10 per widget.
- Process  $p_2$  sends widgets along channel  $c_1$  to  $p_1$ .
- The processes have the initial states shown in the figure on the next slide. It is assumed that  $p_2$  has already received an order for 5 widgets, which it will shortly dispatch to  $p_1$ .

# Example

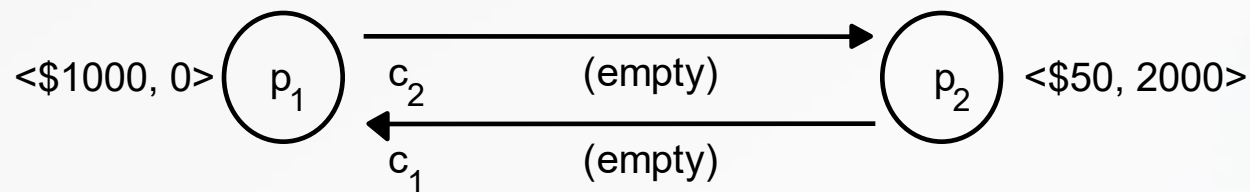


This is the initial state when the algorithm was initiated;  
Before initiation, the process  $p_2$  had received an order for 5 widgets;  
Thus account has \$50

# Example

- Upon request for global snapshot, process  $p_1$  records its state in the actual global state,  $S_0$ , when  $p_1$ 's state (initial) is  $\langle \$1000, 0 \rangle$ .

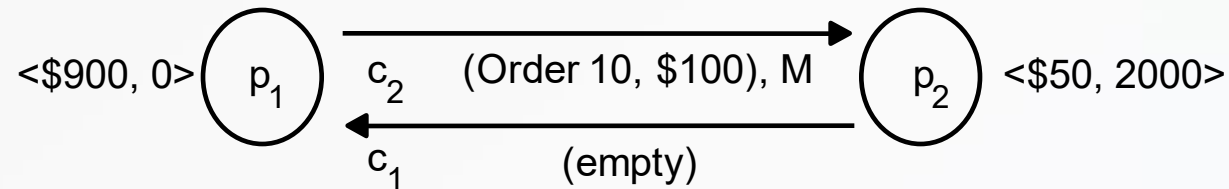
1. Actual Global State  $S_0$



# Example

- Process  $p_1$  then emits a marker message over its outgoing channel  $c_2$  before it sends the next application-level message: (Order 10, \$100) over channel  $c_2$ .
- The system enters actual global state  $S_1$ .

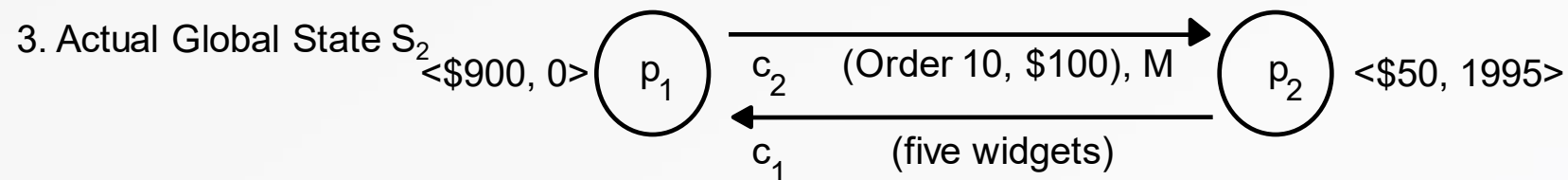
2. Actual Global state  $S_1$





# Example

- Before  $p_2$  receives the marker, it emits an application message (5 widgets) over  $c_1$  in response to  $p_1$ 's previous order, yielding a new actual global state  $S_2$ .



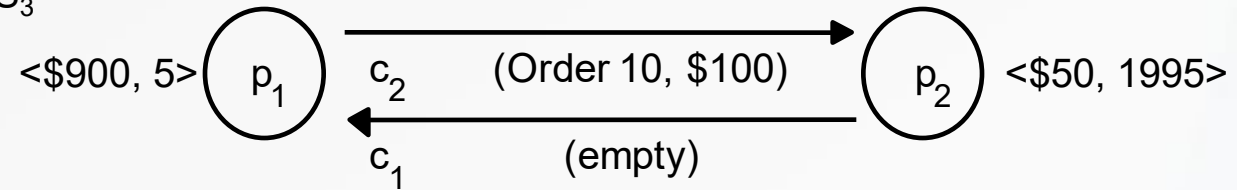
# Example

- Now process  $p_1$  receives  $p_2$ 's message (5 widgets) and  $p_2$  receives the marker.
- $p_2$  records the state of channel  $c_2$  as the empty sequence and follows the marker sending rule.
- $p_2$  records its state as  $\langle \$50, 1995 \rangle$
- $p_2$  sends a marker message over  $c_1$ .
- When process  $p_1$  receives  $p_2$ 's marker message, it records the state of  $c_1$  which is the single message (5 widgets) that it received after it first recorded its state.

# Example

- Final recorded state

4. Actual Global state  $S_3$



# Example

- The sequence of actual global states:
  - $S_0$ :  $p_1$ :  $\langle \$1000, 0 \rangle$ ;  $p_2$ :  $\langle \$50, 2000 \rangle$ ;  $c_1$ :  $\langle \rangle$ ;  $c_2$ :  $\langle \rangle$
  - $S_1$ :  $p_1$ :  $\langle \$900, 0 \rangle$ ;  $p_2$ :  $\langle \$50, 2000 \rangle$ ;  $c_1$ :  $\langle \rangle$ ;  $c_2$ :  $\langle (\text{order } 10, \$100) \rangle$ ;
  - $S_2$ :  $p_1$ :  $\langle \$900, 0 \rangle$ ;  $p_2$ :  $\langle \$50, 1995 \rangle$ ;  $c_1$ :  $\langle \text{five widgets} \rangle$ ;  $c_2$ :  $\langle (\text{order } 10, \$100) \rangle$ ;
  - $S_3$ :  $p_1$ :  $\langle \$900, 5 \rangle$ ;  $p_2$ :  $\langle \$50, 1995 \rangle$ ;  $c_1$ :  $\langle \text{empty} \rangle$ ;  $c_2$ :  $\langle (\text{order } 10, \$100) \rangle$ ;