

CST402 DISTRIBUTED COMPUTING

Module 1: Introduction

CST402


























DISTRIBUTED

COMPUTING

Course Outcome

CO1	Summarize various aspects of distributed computation model and logical time. (Cognitive Knowledge Level: Understand)
CO2	Illustrate election algorithm, global snapshot algorithm and termination detection algorithm. (Cognitive Knowledge Level: Apply)
CO3	Compare token based, non-token based and quorum based mutual exclusion algorithms. (Cognitive Knowledge Level: Understand)
CO4	Recognize the significance of deadlock detection and shared memory in distributed systems. (Cognitive Knowledge Level: Understand)
CO5	Explain the concepts of failure recovery and consensus. (Cognitive Knowledge Level: Understand)
CO6	Illustrate distributed file system architectures. (Cognitive Knowledge Level: Understand)

Mapping of course outcomes with program outcomes

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1												
CO2												
CO3												
CO4												
CO5												
CO6												

Abstract POs defined by National Board of Accreditation

PO#	Broad PO	PO#	Broad PO
PO1	Engineering Knowledge	PO7	Environment and Sustainability
PO2	Problem Analysis	PO8	Ethics
PO3	Design/Development of solutions	PO9	Individual and team work
PO4	Conduct investigations of complex problems	PO10	Communication
PO5	Modern tool usage	PO11	Project Management and Finance
PO6	The Engineer and Society	PO12	Life long learning

Assessment Pattern

Bloom's Category	Continuous Assessment Tests		End Semester Examination Marks (%)
	Test 1 (%)	Test 2 (%)	
Remember	30	30	30
Understand	50	50	50
Apply	20	20	20
Analyze			
Evaluate			
Create			

Mark Distribution

Total Marks	CIE Marks	ESE Marks	ESE Duration
150	50	100	3

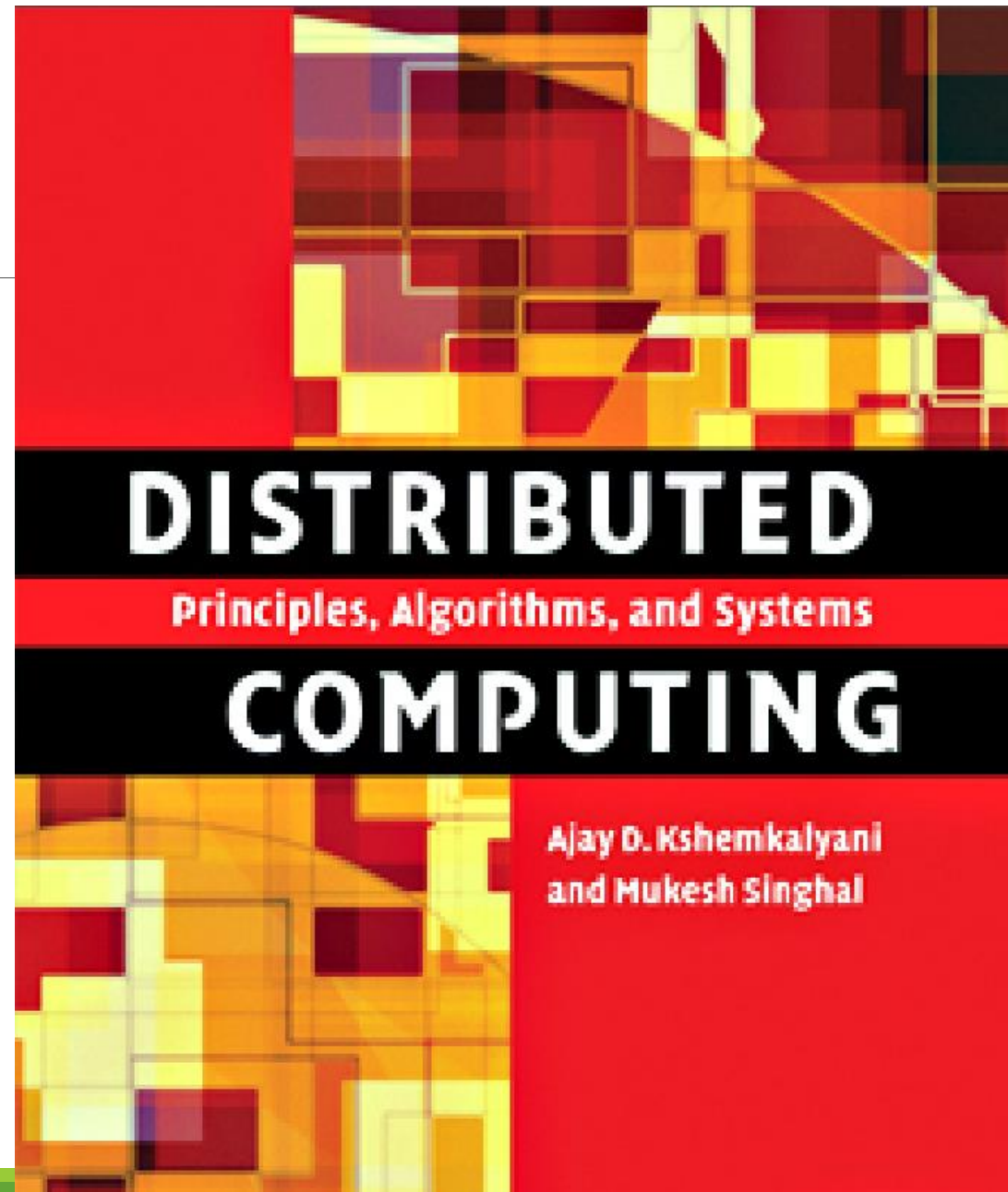
Continuous Internal Evaluation Pattern:

Attendance	10 marks
Continuous Assessment Tests(Average of Internal Tests 1&2)	25 marks
Continuous Assessment Assignment	15 marks

Syllabus- Distributed systems basics and Computation model

Distributed System – Definition, Relation to computer system components, Motivation, Primitives for distributed communication, Design issues, Challenges and applications.

A model of distributed computations – Distributed program, Model of distributed executions, Models of communication networks, Global state of a distributed system, Cuts of a distributed computation, Past and future cones of an event, Models of process communications.



Distributed System

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network

Distributed system has been characterized in one of several ways

1. You know you are using one when **the crash of a computer you have never heard of prevents you from doing work---** prevents losing data in a computer crash
2. 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
3. A collection of independent computers that appears to the users of the system as a single coherent computer
4. A term that describes a wide range of computers, from **weakly coupled systems** such as **wide-area networks**, to **strongly coupled systems** such as **local area networks**, to **very strongly coupled systems** such as **multiprocessor systems**

Features of DS

No common physical clock: This is an important assumption because it introduces the element of “distribution” in the system and gives rise to the inherent asynchrony amongst the processors.

No shared memory This is a key feature that requires message-passing for communication. This feature implies the absence of the common physical clock

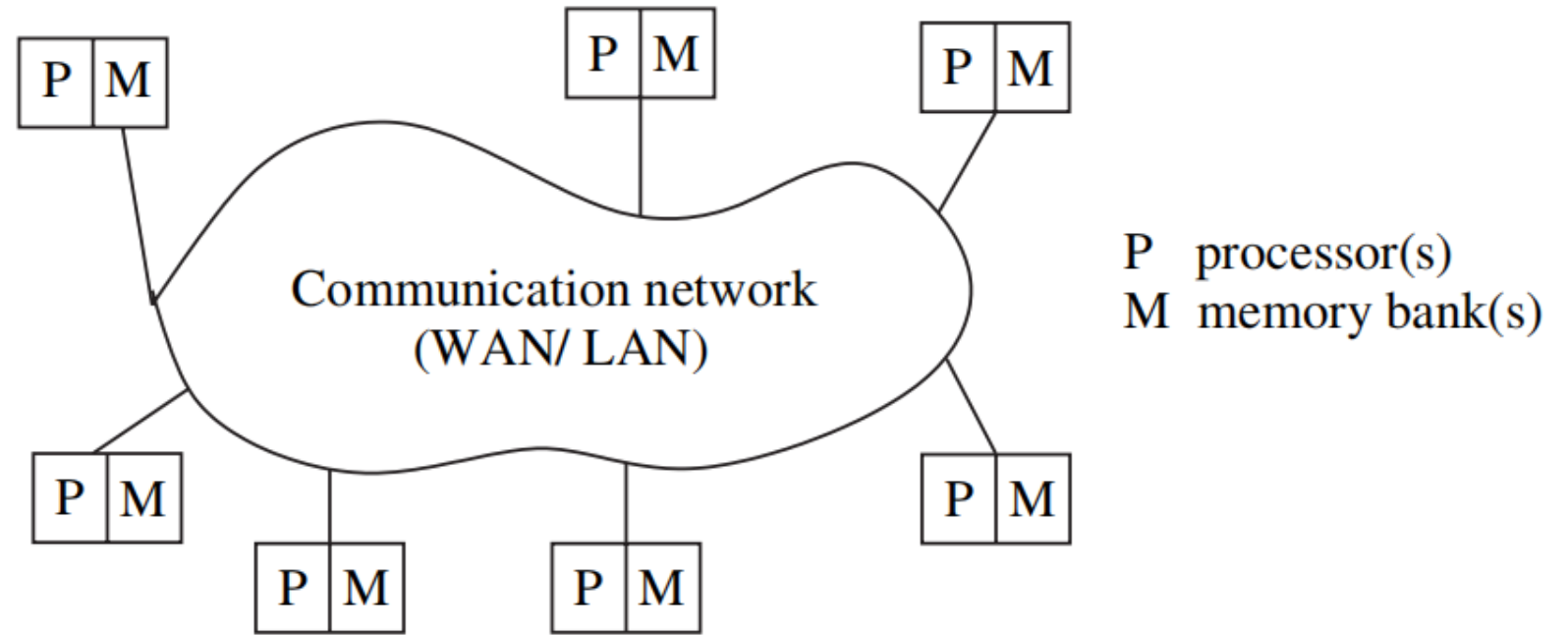
Geographical separation The geographically wider apart that the processors are, the more representative is the system of a distributed system.

- WAN
- NOW/COW(network/cluster of workstations)--- eg, Google search engine

Autonomy and heterogeneity: The processors are “loosely coupled” in that they have different speeds and each can be running a different operating system, **cooperate** with one another by offering services or solving a problem jointly.

Relation to computer system components

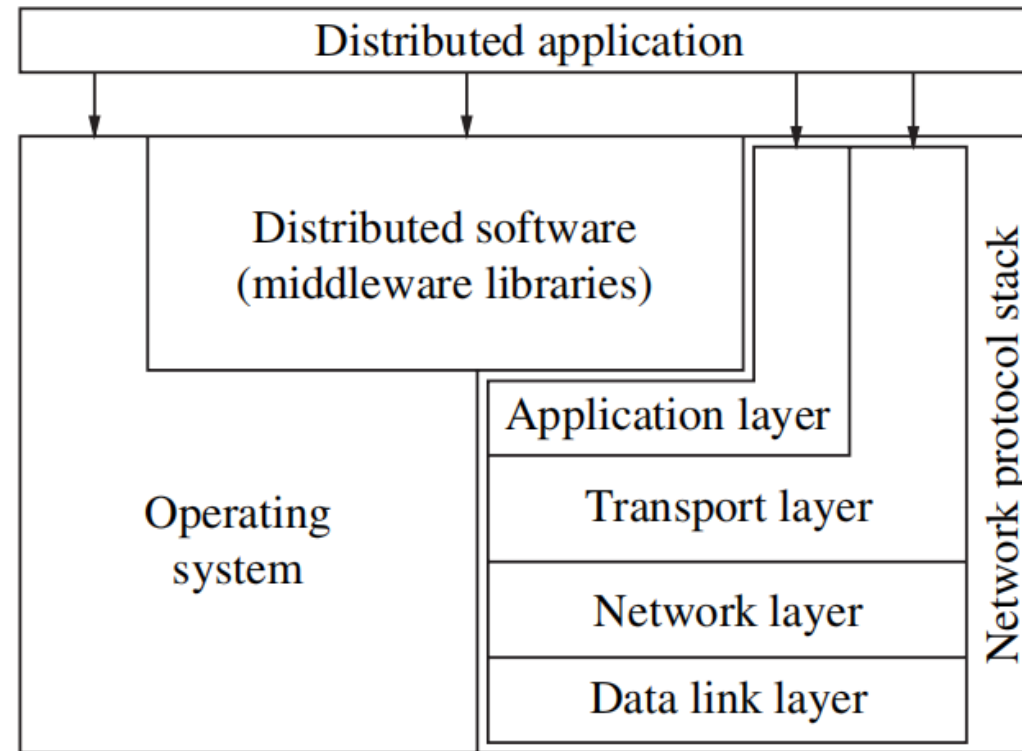
Figure 1.1 A distributed system connects processors by a communication network.



Each computer has a memory-processing unit and the computers are connected by a communication network

Figure 1.2 Interaction of the software components at each processor.

interaction of
software with system
components at each
processor



Extent of
distributed
protocols

relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning

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 distributed system uses a **layered architecture** to break down the complexity of system design.

The **middleware** is the distributed software that **drives the distributed system**, while providing transparency of heterogeneity at the platform level

The middleware layer **does not contain** the traditional application layer functions of the network protocol stack, such as *http, mail, ftp, and telnet*.

Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code.

There exist several libraries to choose from to invoke primitives for the more common functions – such as reliable and ordered multicasting – of the middleware layer

Motivation

1. **Inherently distributed computations**: money transfer in banking, or reaching consensus among parties that are geographically distant
2. **Resource sharing**: 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
3. **Access to geographically remote data and resources**: data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated
4. **Enhanced reliability**: Availability, Integrity, Fault-tolerance
5. **Increased performance/cost ratio**: any task can be partitioned across the various computers in the distributed system
6. **Scalability**
7. **Modularity and incremental expandability**

Module 1: Primitives for distributed communication

Primitives for distributed communication

- Blocking/non-blocking,
- synchronous/asynchronous primitives
- Processor synchrony
- Libraries and standards

Blocking/non-blocking, synchronous/asynchronous primitives

Send()- has at least two parameters, the destination, and the buffer in the user space

Receive()- at least two parameters, the source from which the data is to be received and the user buffer into which the data is to be received

There are two ways of sending data when the *Send* primitive is invoked the

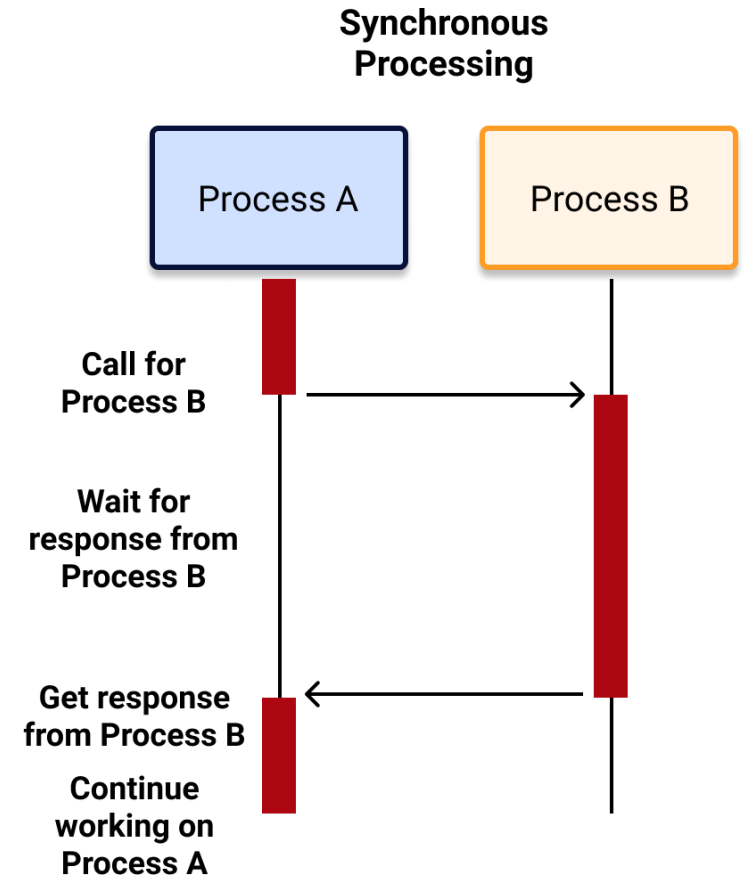
- **buffered option**
 - The *buffered option* which is the standard option copies the data from the user buffer to the kernel buffer.
 - The data later gets copied from the kernel buffer onto the network.
- **unbuffered option**
 - In the *unbuffered option*, the data gets copied directly from the user buffer onto the network.
 - For the *Receive* primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.

Synchronous primitives

A *Send* or a *Receive* primitive is *synchronous* if both the *Send()* and *Receive()* **handshake with each other**.

The processing for the ***Send* primitive** completes only after the invoking processor learns that the other corresponding *Receive* primitive has also been invoked and that **the receive operation has been completed**.

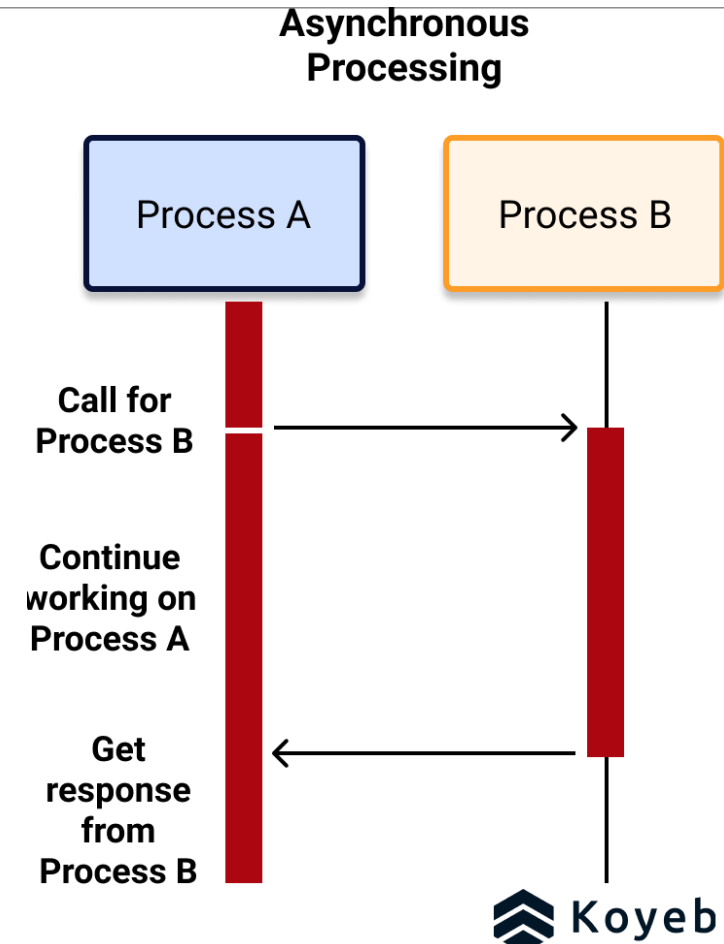
The processing for the ***Receive* primitive** completes when the data to be received is **copied into the receiver's user buffer**.



Asynchronous primitives

A *Send* primitive is said to be *asynchronous* if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.

It does not make sense to define asynchronous Receive primitives.



Blocking primitives

A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.

Non-blocking primitives

A primitive is non-blocking if control returns back to the invoking process **immediately after invocation**, even though the operation has not completed.

For a **non-blocking *Send***, control returns to the process even before the data is copied out of the user buffer.

For a **non-blocking *Receive***, control returns to the process even before the data may have arrived from the sender.

Non Blocking primitive - handle

For **non-blocking primitives**, a return parameter on the primitive call returns a **system-generated handle** which can be later used to check the status of completion of the call.

The process can check for the completion of the call in two ways.

1. First, it can **keep checking** (in a loop or periodically) if the handle has been **flagged or posted**.
2. Second, it can issue a **Wait** with a **list of handles as parameters**.

Blocking wait () on non blocking primitives

The *Wait* call usually blocks until one of the parameter handles is posted.

Figure 1.7 A non-blocking *send* primitive. When the *Wait* call returns, at least one of its parameters is posted.

<i>Send(X, destination, handle_k)</i>	<i>// handle_k is a return parameter</i>
<i>...</i>	
<i>...</i>	
<i>Wait(handle₁, handle₂, ..., handle_k, ..., handle_m)</i>	<i>// Wait always blocks</i>

Blocking wait () on non blocking primitives and posting

- If at the time that *Wait()* is issued, the processing for the primitive has completed, the *Wait* returns immediately
- The completion of the processing of the primitive is detectable by checking the value of $handle_k$.
- If the processing of the primitive **has not completed**, the ***Wait* blocks** and waits for a signal to wake it up.
- When the processing for the **primitive completes**,
 - the communication subsystem software **sets the value of** $handle_k$
 - **and wakes up** (signals) any process with a *Wait* call blocked on this $handle_k$.

This is called ***posting*** the **completion of the operation**.

Four Versions of the Send primitive

1. synchronous blocking,
2. synchronous non-blocking,
3. asynchronous blocking, and
4. asynchronous non-blocking



Two Versions of the receive primitive

1. Blocking synchronous
2. Non-blocking synchronous

Timing Diagram

Here, three time lines are shown for each process:

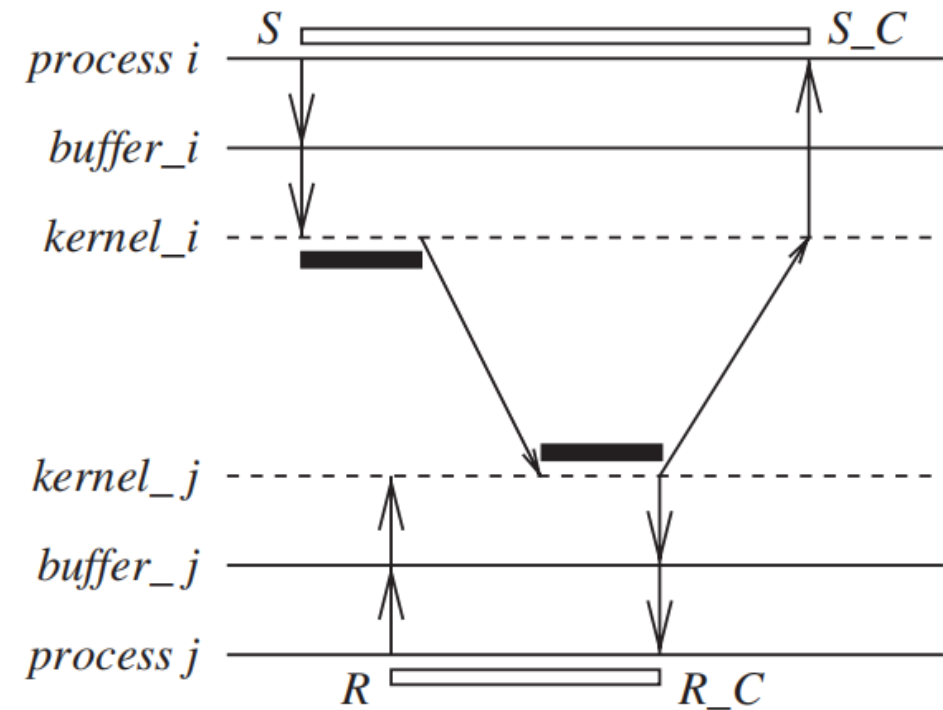
- (1) for the process execution,
- (2) for the user buffer from/to which data is sent/received, and
- (3) for the kernel/communication subsystem.

	Duration to copy data from or to user buffer		
	Duration in which the process issuing send or receive primitive is blocked		
<i>S</i>	<i>Send</i> primitive issued	<i>S_C</i>	processing for <i>Send</i> completes
<i>R</i>	<i>Receive</i> primitive issued	<i>R_C</i>	processing for <i>Receive</i> completes
<i>P</i>	The completion of the previously initiated nonblocking operation		
<i>W</i>	Process may issue <i>Wait</i> to check completion of nonblocking operation		

Blocking synchronous Send

The data gets copied from the user buffer to the kernel buffer and is then sent over the network.

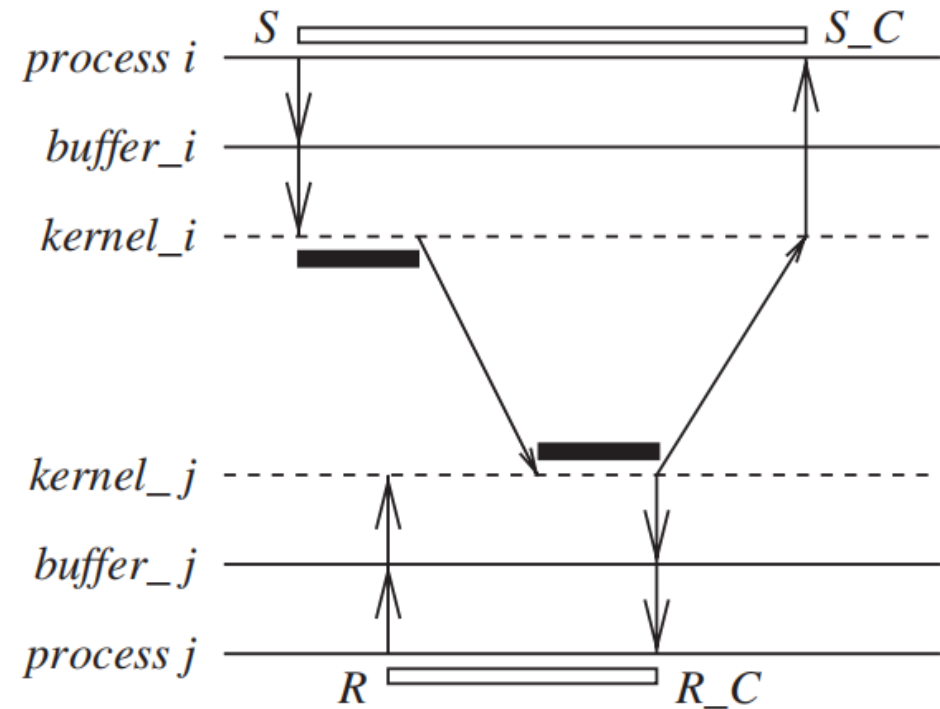
After the data is copied to the receiver's system buffer and a *Receive* call has been issued, an acknowledgement back to the sender causes control to return to the process that invoked the *Send* operation and completes the *Send*



Blocking Receive

The *Receive* call blocks until the data expected arrives and is written in the specified user buffer.

Then control is returned to the user process.



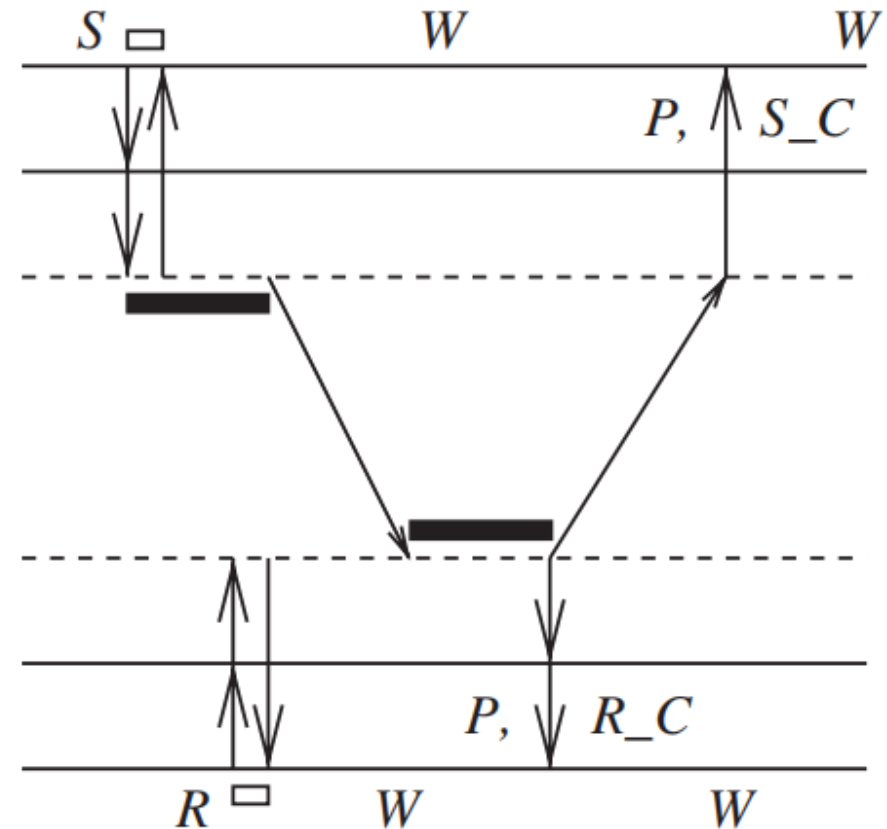
Non-blocking synchronous Send

Control returns back to the invoking process **as soon as the copy of data from the user buffer to the kernel buffer is initiated**.

A parameter in the non-blocking call also gets set with the **handle of a location that the user process can later check for the completion of the synchronous send operation**.

The location gets posted **after an acknowledgement returns from the receiver**

The user process can keep checking for the completion of the non-blocking synchronous *Send* by **testing the returned handle**, or it can invoke the blocking *Wait* operation on the returned handle

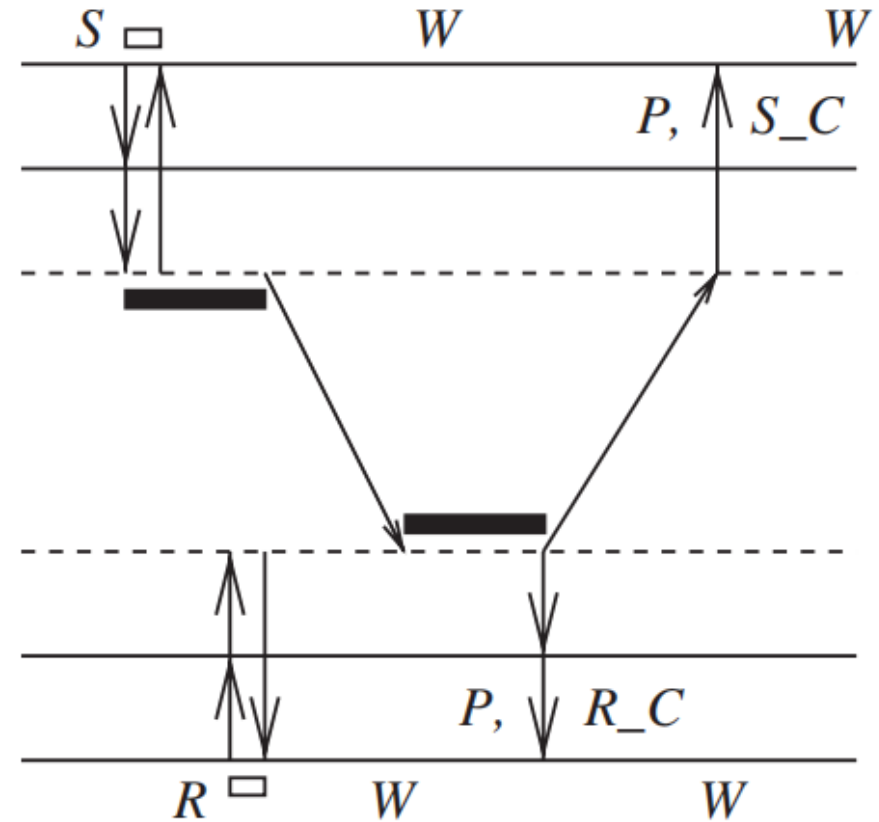


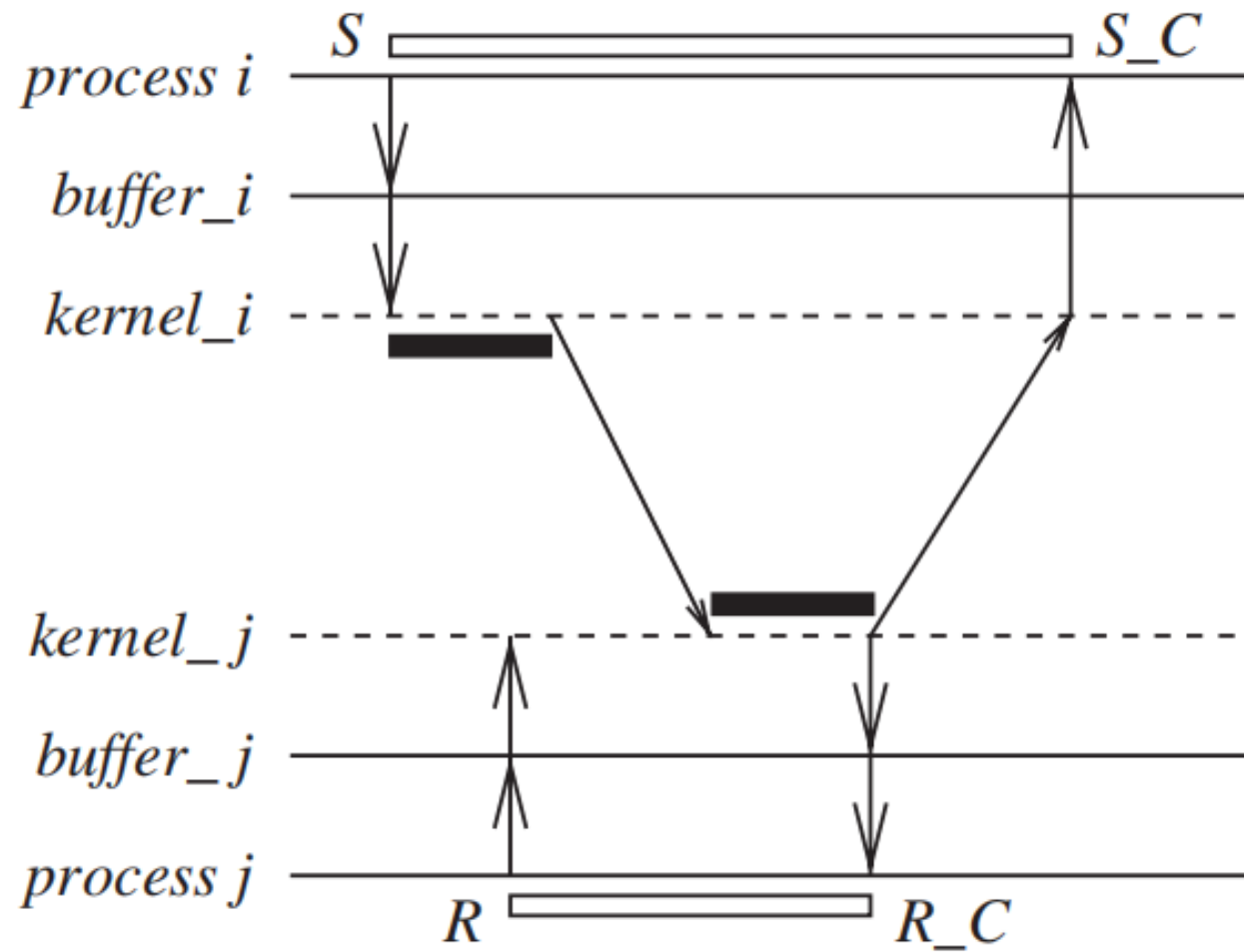
Non-blocking Receive

The *Receive* call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking *Receive* operation.

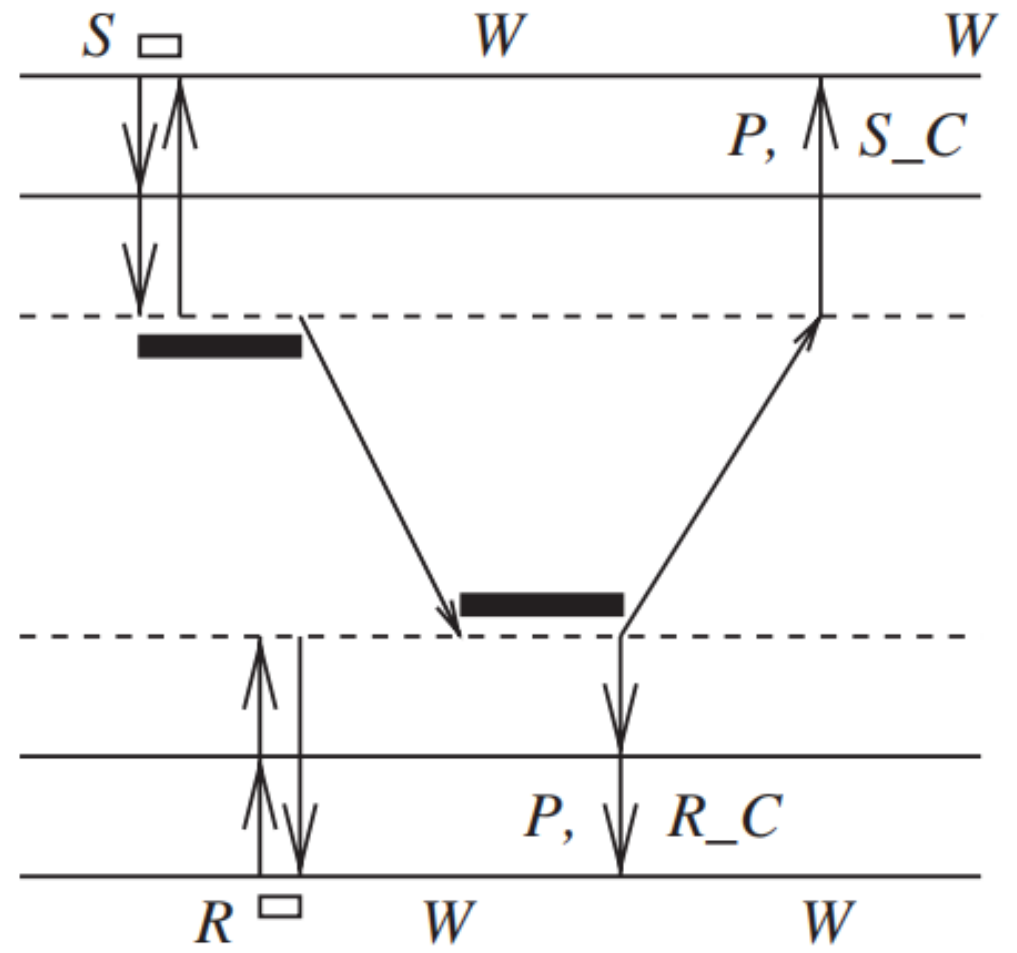
This location gets posted by the kernel after the expected data arrives and is copied to the user-specified buffer.

The user process can check for the completion of the non-blocking *Receive* by invoking the *Wait* operation on the returned handle.





(a) Blocking sync. *Send*, blocking *Receive*

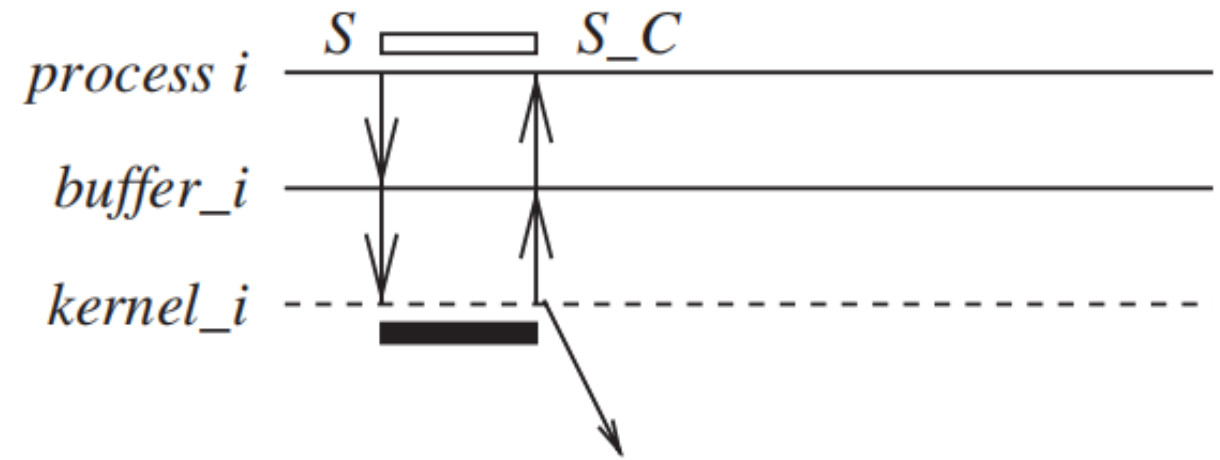


(b) Nonblocking sync. *Send*, nonblocking *Receive*

Blocking asynchronous Send

The user process that invokes the **Send** is **blocked** until the data is copied from the user's buffer to the kernel buffer.

For the **unbuffered** option, the user process that invokes the **Send** is **blocked** until the data is copied from the user's buffer to the network.



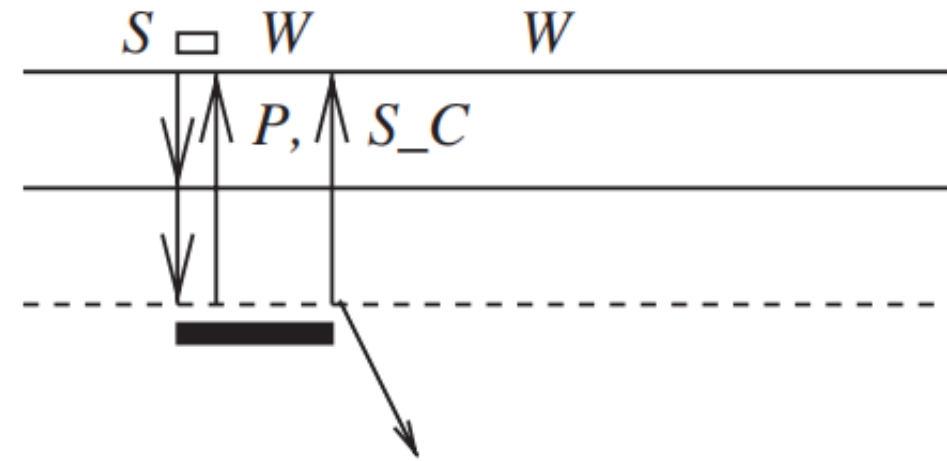
Non-blocking asynchronous Send

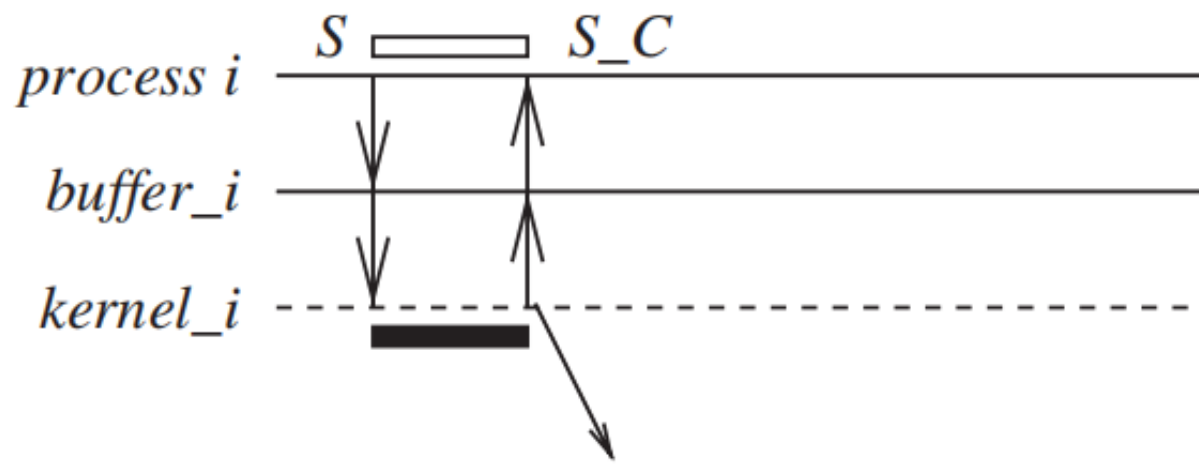
The user process that invokes the *Send* is **blocked** until the transfer of the data from the user's buffer to the kernel buffer is **initiated**.

Control returns to the user process as soon as this **transfer is initiated**, and a parameter in the non-blocking call also gets set with the handle of a location that the user process can check later using the **Wait operation** for the **completion of the asynchronous Send operation**.

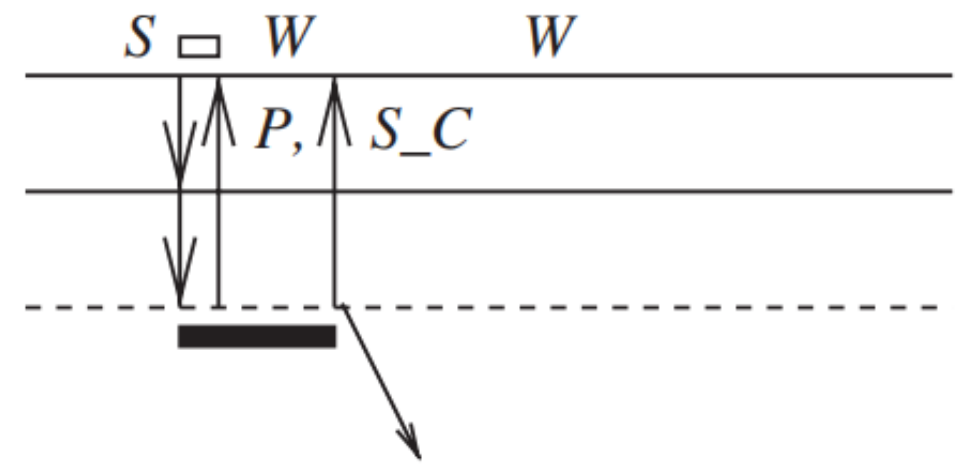
The asynchronous *Send* completes when the data has been copied out of the user's buffer.

The checking for the completion may be necessary if the user wants to **reuse the buffer from which the data was sent**.





(c) Blocking async. *Send*



(d) Non-blocking async. *Send*

■ Duration to copy data from or to user buffer

□ Duration in which the process issuing send or receive primitive is blocked

S *Send* primitive issued

S_C processing for *Send* completes

R *Receive* primitive issued

R_C processing for *Receive* completes

P The completion of the previously initiated nonblocking operation

W Process may issue *Wait* to check completion of nonblocking operation

A synchronous *Send* is easier to use from a programmer's perspective because the handshake between the *Send* and the *Receive* makes the communication appear instantaneous

- “instantaneity” is, of course, only an illusion
- The *Receive* may not get issued until much after the data arrives at P_j , in which case the data arrived would have to be buffered in the system buffer at P_j and not in the user buffer.
- At the same time, the sender would remain blocked.
- Thus, a synchronous *Send* lowers the efficiency within process P_i .

The **non-blocking asynchronous *Send*** is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the *Send*.

- The non-blocking synchronous *Send* also avoids the potentially large delays for handshaking, particularly when the receiver has not yet issued the *Receive* call.

The **non-blocking *Receive*** is useful when a large data item is being received and/or when the sender has not yet issued the *Send* call,

- because it allows the process to perform other instructions in parallel with the completion of the *Receive*.
- If the data has already arrived, it is stored in the kernel buffer, and it may take a while to copy it to the user buffer specified in the *Receive* call.

For **non-blocking calls**, however, the burden on the programmer increases because he or she has to keep track of the completion of such operations in order to meaningfully reuse (write to or read from) the user buffers

Primitives for distributed communication

Blocking/non-blocking, synchronous/asynchronous primitives

Processor synchrony

Libraries and standards

Processor synchrony

Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.

As this synchrony is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a *step*, the processors are synchronized.

This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

Design issues and challenges

3 Categories:

1. **System perspective**, having a greater component related to systems design and operating systems design, or
2. **Algorithmic challenges**, having a greater component related to algorithm design, or
3. **Applications of Distributed computing and new challenges:**
emerging from recent technology advances and/or driven by new applications

Distributed systems challenges from a system perspective

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

1. Communication
2. Processes
3. Naming
4. Synchronization
5. Data storage and access
6. Consistency and replication
7. Fault tolerance
8. Security
9. Applications Programming Interface (API) and transparency
10. Scalability and modularity

1. Communication

- Designing mechanisms for communication among the processes in the network
- Some example mechanisms :
 - remote procedure call (RPC),
 - remote object invocation (ROI),
 - message-oriented communication versus stream-oriented communication.

2. Processes

Some of the issues involved are:

- management of processes and threads at clients/servers;
- code migration
- design of software and mobile agents.

3. Naming

- need *easy to use* and *robust* schemes for *names, identifiers, and addresses* ->for locating resources and processes in a transparent and scalable manner
- Naming in mobile systems provides additional challenges because **naming cannot easily be tied to any static geographical topology.**

4. Synchronization

- Mechanisms for *synchronization* or *coordination* among the processes needed
- Mutual exclusion -the classical example of synchronization,
- Different forms of synchronization needed
 - leader election ,
 - synchronizing physical clocks
 - devising logical clocks that capture the essence of the passage of time
 - global state recording algorithms etc.

5. Data storage and access

- Schemes for data storage and accessing the data in a fast and scalable manner across the network are important for efficiency.
- *File system design* have to be reconsidered in the setting of a distributed system.

6. Consistency and replication

- Replication ->To avoid bottlenecks, provide fast access to data, and to provide scalability
- Issues:
 - managing the replicas,
 - consistency among the replicas/caches in a distributed setting
- E.g. issue is deciding the level of granularity (i.e., size) of data access.

7. Fault tolerance

- Maintaining **correct and efficient operation** in spite of failures of **links, nodes, and processes**
- Some of the mechanisms to provide fault-tolerance are:
 - **Process resilience,**
 - **reliable communication,**
 - **distributed commit, checkpointing and recovery,**
 - **agreement and consensus, failure detection, and self-stabilization**

8. Security

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

9. Applications Programming Interface (API) and transparency

Transparency deals with hiding the implementation policies from the user, and can be classified as follows

1. *Access transparency* hides differences in data representation on different systems and provides uniform operations to access system resources.
2. *Location transparency* makes the locations of resources transparent to the users.
3. *Migration transparency* allows relocating resources without changing names.
4. *Relocation transparency*: The ability to relocate the resources as they are being accessed is.
5. *Replication transparency* does not let the user become aware of any replication.
6. *Concurrency transparency* deals with masking the concurrent use of shared resources for the user.
7. *Failure transparency* refers to the system being reliable and fault-tolerant.

10. 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.

Algorithmic challenges in distributed computing

- ❑ Designing useful execution models and frameworks
- ❑ Dynamic distributed graph algorithms and distributed routing algorithms
- ❑ Time and global state in a distributed system
- ❑ Synchronization/coordination mechanisms
- ❑ Group communication, multicast, and ordered message delivery
- ❑ Monitoring distributed events and predicates
- ❑ Distributed program design and verification tools
- ❑ Debugging distributed programs
- ❑ Data replication, consistency models, and caching

Designing useful execution models and frameworks

- The *interleaving model* and *partial order model* are two widely adopted models of *distributed system executions*.
- Useful for operational reasoning , design of distributed algorithms
- The *input/output automata model* and the *TLA (temporal logic of actions)*- other egs.

Dynamic distributed graph algorithms and distributed routing algorithms

- The distributed system is modeled as a **distributed graph**,
- The **graph algorithms** form the building blocks for → higher level communication, data dissemination, object location, and object search functions.
- The algorithms need to deal with ***dynamically changing graph characteristics***, such as to model *varying link loads* in a routing algorithm.
- The efficiency of these algorithms impacts user-perceived latency ,traffic and load or congestion in the network
- The design of efficient distributed graph algorithms ->paramount importance

Time and global state in a distributed system

The challenges pertain to providing accurate **physical time**, and to providing a variant of time, called **logical time**

Logical time is relative time, and eliminates the overheads of providing physical time for applications where physical time is not required. More importantly, logical time can

- i. capture the logic and inter-process dependencies within the distributed program, and also
- ii. track the relative progress at each process.

It is not possible for any one process to directly observe a meaningful global state across all the processes, without using extra state-gathering effort which needs to be done in a coordinated manner

Synchronization/coordination mechanisms

Synchronization is essential to overcome the limited observation of the system state from the viewpoint of any one process.

Overcoming this limited observation is necessary for taking any actions that would impact other processes.

The synchronization mechanisms can also be viewed as **resource management** and **concurrency management** mechanisms to streamline the behavior of the processes that would otherwise act independently.

Problems Requiring Synchronization

Physical clock synchronization

Leader election

Mutual exclusion

Deadlock detection and resolution

Termination detection

Garbage collection

Leader election

All the processes need to agree on which process will play the role of a distinguished process – called a **leader process**.

A leader is necessary even for many distributed algorithms because there is often some asymmetry as in initiating some action like a **broadcast** or collecting the **state of the system**, or in “**regenerating**” a token that gets “**lost**” in the system

Group communication, multicast, and ordered message delivery

A **group** is a collection of processes that share a common context and collaborate on a common task within an application domain.

Specific algorithms need to be designed to enable efficient group **communication** and group **management** wherein processes can join and leave groups dynamically, or even fail.

When multiple processes send messages concurrently, different recipients may receive the messages in different orders, possibly violating the semantics of the distributed program.

Hence, **formal specifications of the semantics of ordered delivery** need to be formulated, and then implemented.

Monitoring distributed events and predicates

Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state, and are useful for applications such as **debugging, sensing the environment, and in industrial process control.**

On-line algorithms for monitoring such predicates are hence important.

An important paradigm for monitoring distributed events is that of *event streaming*, wherein **streams of relevant events reported from different processes are examined collectively to detect predicates.**

Typically, the specification of such predicates uses physical or logical time relationships.

Distributed program design and verification tools

Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering.

Designing mechanisms to achieve these design and verification goals is a challenge.

Debugging distributed programs

Debugging sequential programs is hard; debugging distributed programs is that much harder because of the **concurrency in actions** and the ensuing uncertainty due to **the large number of possible executions defined by the interleaved concurrent actions**.

Adequate debugging mechanisms and tools need to be designed to meet this challenge.

Data replication, consistency models, and caching

Fast access to data and other resources requires them to be replicated in the distributed system.

Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies.

Additionally, placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.

Distributed Shared Memory abstraction

- Simplifies task of a programmer.
- Only read and write , instead of communication primitives
- More expensive

Reliable and fault tolerant DS

- Consensus algorithms : correctly functioning processes to reach consensus/agreement , in the presence of malicious processes
- Replication and replica management
- Voting and quorum systems
- Distributed databases and distributed commit
- Self stabilizing systems
- Checkpointing and recovery algorithms
- Failure detectors

Applications of distributed computing and newer challenges

1. Mobile systems
2. Sensor networks
3. Ubiquitous or pervasive computing
4. Peer-to-peer computing
5. Publish-subscribe, content distribution, and multimedia
6. Distributed agents
7. Distributed data mining
8. Grid computing
9. Security in distributed systems

1. Mobile systems

- Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium.
- Characteristics of communication are different;
- Set of problems such as
 - i. routing,
 - ii. location management,
 - iii. channel allocation,
 - iv. localization and position estimation, and
 - v. the overall management of mobility

Mobile systems – 2 architectures

- **Base-station approach**, as the *cellular approach*
 - a **cell** which is the geographical region within range of a static but powerful base transmission station **is associated with that base station**
 - All mobile processes in the cell communicate with rest of the system via base station
- **ad-hoc network approach**
 - no base station
 - All responsibility for communication is distributed among the mobile nodes

2. Sensor networks

A sensor is a processor with an **electro-mechanical interface** that is capable of sensing physical parameters, such as temperature, velocity, pressure, humidity, and chemicals

Sensors may be mobile or static;

sensors may communicate wirelessly, although they may also communicate across a wire when they are statically installed.

Sensors may have to self-configure to form an ad-hoc network, which introduces a whole new set of challenges, such as position estimation and time estimation

3. Ubiquitous or pervasive computing

- Processors are embedded in and perform application functions in the background .**intelligent home, smart workplace** etc . Egs.
- Ubiquitous systems are distributed systems, wireless communication and sensor and actuator mechanisms
- They can be self-organizing and network-centric, while also being resource constrained
- Characterized as having many small processors operating collectively in a dynamic ambient network.
- The processors may be connected to more powerful networks and processing resources in the background for processing and collating data.

4. Peer-to-peer computing

- Represents computing over an application layer network , all interactions among the processors are at a “peer” level, without any hierarchy among the processors.
- Self-organizing, and may or may not have a regular structure to the network
- No central directories for name resolution and object lookup
- Challenges: Object storage mechanisms, efficient object look up and retrieval, dynamic reconfiguration, replication strategies etc.

5. Publish-subscribe, content distribution, and multimedia

In a dynamic environment where the information constantly fluctuates (**varying stock prices**), there needs to be:

- i. an efficient mechanism for **distributing** this information (publish),
- ii. an efficient mechanism to allow end users to **indicate interest** in receiving specific kinds of information (subscribe), and
- iii. an efficient mechanism for **aggregating large volumes** of published information and **filtering** it as per the user's subscription filter

Content distribution: specific info -> distributed to interested processes

multimedia data is usually very large and information-intensive, requires compression, and special synchronization during storage and playback

6. Distributed agents

Software processes/robots that can move around the system, do specific tasks for which specifically programmed

Agents collect, process information, exchange information with other agents

Agents cooperate as in an ant colony, can also have friendly competition, as in a free market economy.

Challenges in distributed agent systems include **coordination** mechanisms among the agents, controlling the **mobility** of the agents, and their software design and interfaces.

Research in agents is **inter-disciplinary**: spanning artificial intelligence, mobile computing, economic market models, software engineering, and distributed computing.

7. Distributed data mining

- Datamining->examine large amounts of data-detect patterns/trends,to mine/extract useful info
- Mining ->db+AI techniques to a data repository
- The data is necessarily distributed and cannot be collected in a single repository, as in **banking applications**,or in **atmospheric weather prediction**
- Efficient distributed datamining algorithms required

8. Grid computing

Grid Computing is a subset of distributed computing, where a virtual supercomputer comprises **machines on a network connected by some bus**, mostly Ethernet or sometimes the Internet.

It can also be seen as a form of Parallel Computing where instead of many CPU cores on a single machine, it **contains multiple cores spread across various locations**.

Many challenges in making grid computing a reality include:

- scheduling jobs in such a distributed environment,
- a framework for implementing quality of service and real-time guarantees,
- Security of individual machines as well as of jobs being executed in this setting.

9. Security in distributed systems

The traditional challenges of security in a distributed setting include:

- **confidentiality** (ensuring that only authorized processes can access certain information),
- **authentication** (ensuring the source of received information and the identity of the sending process), and
- **availability** (maintaining allowed access to services despite malicious actions).

The goal is to meet these challenges with efficient and scalable solutions.

These basic challenges have been addressed in traditional distributed settings.

A model of distributed computations

- ❑ A distributed system consists of a **set of processors** that are connected by a **communication network**.
- ❑ The communication network provides the facility of **information exchange** among processors.
- ❑ The **communication delay is finite** but **unpredictable**.
- ❑ The processors **do not share a common global memory** and communicate solely by passing messages over the communication network.
- ❑ There is no **physical global clock in the system** to which processes have instantaneous access.
- ❑ The communication medium **may deliver messages out of order**, messages may be **lost, garbled, or duplicated** due to timeout and retransmission, processors may fail, and communication links may go down.
- ❑ The system can be modeled as **a directed graph** in which vertices represent the **processes** and edges represent **unidirectional communication channels**.

A distributed application runs as a collection of processes on a distributed system

A distributed program

A distributed program is composed of a **set of n asynchronous processes** $p_1, p_2, \dots, p_i, \dots, p_n$ that communicate by **message passing** over the **communication network**.

- we assume that each process is running on a different processor

The **processes do not share a global memory** and communicate solely by passing messages

- C_{ij} : denote the **channel** from process p_i to process p_j
- m_{ij} : denote a **message** sent by p_i to p_j .

A distributed program contd...

- Communication delay is **finite and unpredictable**
- Processes **do not share a global clock** that is instantaneously accessible to these processes
- Process execution and message transfer are **asynchronous**
 - a process may execute an action **spontaneously**
 - a process sending a message **does not wait for the delivery** of the message to be **complete**

A distributed program contd...

- The **global state** of a distributed computation is composed of the **states of the processes and the communication channels**
 - The **state of a process** is characterized by the *state of its local memory* and depends upon the *context*.
 - The **state of a channel** is characterized by the *set of messages in transit* in the channel.

A model of distributed executions

- The **execution of a process** consists of a sequential execution of its **actions**.
- The actions are atomic and the actions of a process are modeled as three types of events,
 - internal events,
 - message send events, and
 - Message receive events
- For a message m , let $\text{send}(m)$ and $\text{rec}(m)$ denote its send and receive events

Model of distributed executions contd..

- The **occurrence of events** changes the states of respective processes and channels->**transitions in the global system state**.
- An **internal event** changes the **state of the process at which it occurs**
- A **send event** (or a receive event) **changes the state of the process that sends** (or receives) the message **and the state of the channel** on which the message is sent (or received)
- An **internal event** only affects the process at which it occurs

Model of distributed executions contd..

Let e_i^x denote the x_{th} event at process p_i

The events at a process are **linearly ordered by their order of occurrence.**

The execution of process p_i produces a sequence of events $e_i^1, e_i^2, \dots, e_i^x, e_i^{x+1}, \dots$ and is denoted by \mathcal{H}_i :

$$\mathcal{H}_i = (h_i, \rightarrow_i),$$

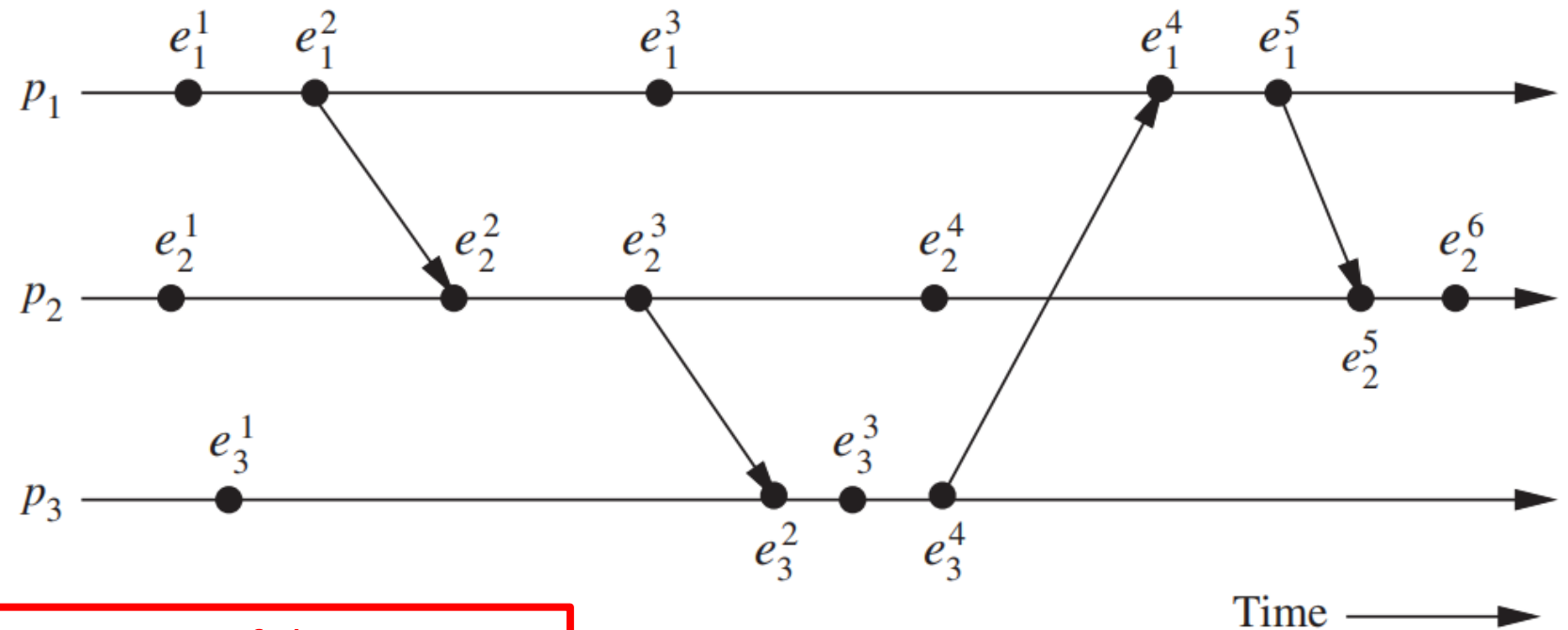
where h_i is the set of events produced by p_i and binary relation \rightarrow_i defines a linear order on these events. Relation \rightarrow_i expresses causal dependencies among the events of p_i .

The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process. A relation \rightarrow_{msg} that captures the causal dependency due to message exchange, is defined as follows. For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m).$$

Space–Time Diagram of a Distributed Execution Involving Three Processes

Figure 2.1 The space–time diagram of a distributed execution.



A horizontal line represents the progress of the process;
a dot indicates an event;
a slant arrow indicates a message transfer.

Causal precedence relation

The execution of a distributed application results in a **set of distributed events** produced by the processes

Let $H = \bigcup_i h_i$ denote the set of events executed in a distributed computation
causal dependencies between events in the distributed execution: define a binary relation on the set H , denoted as \rightarrow

$$\forall e_i^x, \forall e_j^y \in H, \quad e_i^x \rightarrow e_j^y \Leftrightarrow \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases}$$

The causal precedence relation induces **an irreflexive partial order** on the events of a distributed computation that is denoted as $\mathcal{H} = (H, \rightarrow)$

For example $e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$

relation \rightarrow is Lamport's “happens before” relation

For any two events dependent on event e_i and e_j , if $e_i \rightarrow e_j$, then event e_j is **directly or transitively dependent on event e_i** ;

graphically, it means that there **exists a path consisting of message arrows and process-line segments in the space–time diagram that starts at e_i and ends at e_j** .

relation \rightarrow denotes flow of information in a distributed computation and **$e_i \rightarrow e_j$ dictates that all the information available at e_i is potentially accessible at e_j** .

Eg: event e_2^6 has the knowledge of all other events shown in the figure

For any two events e_i and e_j , $e_i \not\rightarrow e_j$ denotes

- event e_j does not directly or transitively dependent on event e_i
- event e_i does not causally affect event e_j .
- Event e_j is not aware of the execution of e_i or any event executed after e_i on the same process.
- For example $e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$
- two rules:
 - for any two events e_i and e_j , $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$
 - for any two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.
- For any two events e_i and e_j , if $e_i \not\rightarrow e_j$ and $e_j \not\rightarrow e_i$, then events e_i and e_j are said to be **concurrent** and the relation is denoted as $e_i \parallel e_j$
- relation \parallel is not transitive
$$(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$$

Causal precedence relation contd..

for any two events e_i and e_j in a distributed execution,

- $e_i \rightarrow e_j$ or
- $e_j \rightarrow e_i$, or
- $e_i \parallel e_j$

Logical vs. Physical Concurrency

In a distributed computation, two events are **logically concurrent** if and only if they do not **causally affect each other**.

Physical concurrency, on the other hand, has a meaning that the **events occur at the same instant in physical time**.

Two or more events may be **logically concurrent** even though they do not occur at the same instant in physical time.

However, if processor speed and message delays would have been different, the execution of these events could have very well coincided in physical time.

Whether **a set of logically concurrent events coincide in the physical time or not, does not change the outcome of the computation**.

Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, we can assume that these events occurred at the same instant in physical time.

Models of communication networks

3 models of the service provided by communication networks

1. **FIFO (first-in, first-out) Model:**

- Each channel acts like a **FIFO Message Queue**
- Message ordering is preserved by a channel

2. **Non-FIFO Model:**

- a channel acts like a **set** in which the sender process adds messages and the receiver process removes messages from it in a random order

3. **Causal ordering Model**

Models of communication networks contd...

Causal ordering:

- based on Lamport's "happens before" relation
- A system that supports the causal ordering model satisfies the following property

CO: For any two messages m_{ij} and m_{kj} , if $send(m_{ij}) \longrightarrow send(m_{kj})$,
then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

- this property ensures that **causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.**

$$CO \subset FIFO \subset \text{Non-FIFO}$$

Models of communication networks contd...

- ❑ Causal ordering model is useful in developing **distributed algorithms**.
- ❑ simplifies the design of distributed algorithms ->provides a built-in synchronization
- ❑ For example, in replicated database systems, it is important that every process responsible for updating a replica receives the updates in the same order to maintain database consistency.
- ❑ Without causal ordering, each update must be checked to ensure that database consistency is not being violated. Causal ordering eliminates the need for such checks.

Global state of a distributed system

The **global state** of a distributed system is a **collection of the local states** of its components->**processes and the communication channels**

The **state of a process** -> defined by contents of *processor registers, stacks, local memory*, etc. and depends on the *local context* of the distributed application

The **state of a channel** -> the set of messages in transit in the channel

The **occurrence of events** changes the states of respective processes and channels, thus causing **transitions in global system state**. For eg,

- an **internal event** changes the state of the process at which it occurs.
- A **send event** (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received)

Let LS_i^x denote the **state of process** p_i after the occurrence of event e_i^x and before the event e_i^{x+1}

LS_i^0 denotes the initial state of process p_i

LS_i^x is a **result of the execution of all the events executed by process p_i till e_i^x**

$send(m) \leq LS_i^x$ denote the fact that $\exists y: 1 \leq y \leq x :: e_i^y = send(m)$

$rec(m) \not\leq LS_i^x$ denote the fact that $\forall y: 1 \leq y \leq x :: e_i^y \neq rec(m)$

Let $SC_{ij}^{x,y}$ denote the state of a channel C_{ij} defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq LS_i^x \wedge rec(m_{ij}) \not\leq LS_j^y\}.$$

Global state of a distributed system contd...

Let e_i^x denote the x_{th} event at process p_i

Let LS_i^x denote the **state of process** p_i after the occurrence of event e_i^x and before the event e_i^{x+1}

Let e_i^x denote the x_{th} event at process p_i

Global state

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

$$GS = \{\underbrace{\bigcup_i LS_i^{x_i}}_{\text{States}}, \underbrace{\bigcup_{j,k} SC_{jk}^{y_j, z_k}}_{\text{Channels}}\}.$$

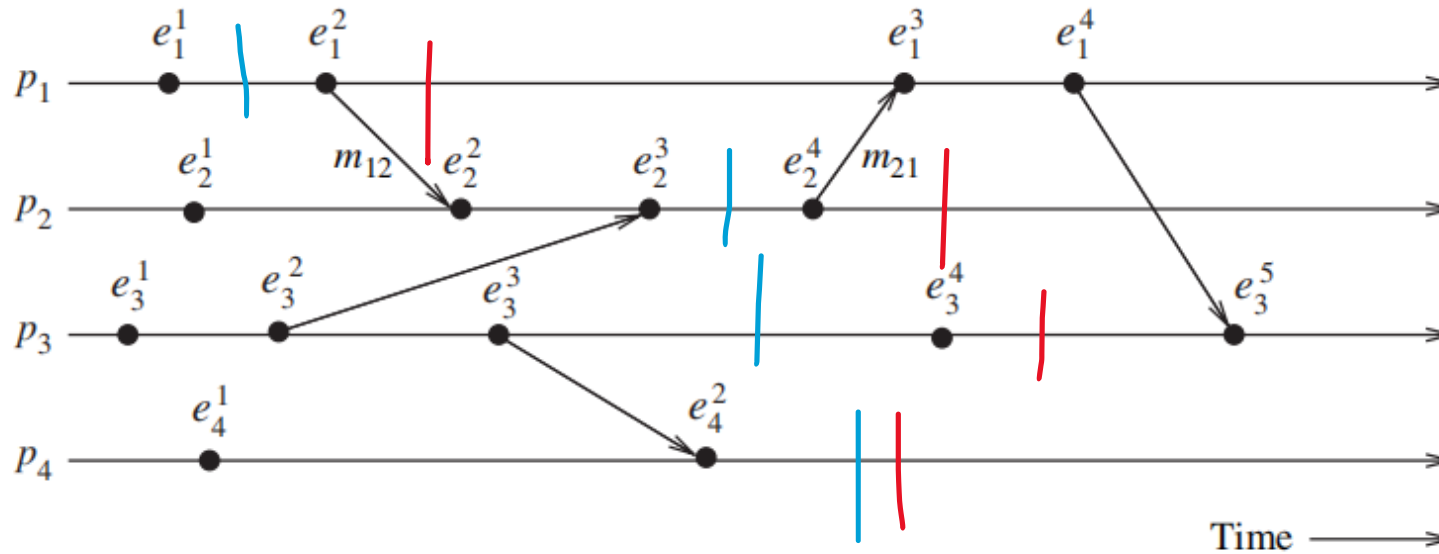
- Even if the state of all the components in a distributed system has not been recorded at the same instant, such a state will be meaningful **provided every message that is recorded as received is also recorded as sent.**
- Basic idea is that **an effect should not be present without its cause.**
- A **message cannot be received if it was not sent**; that is, the state should not violate causality.
- Such states are called **consistent global states** and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

Global state contd...

A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$ is a *consistent global state* iff it satisfies the following condition:

$$\forall m_{ij} : send(m_{ij}) \not\leq LS_i^{x_i} \Rightarrow m_{ij} \notin SC_{ij}^{x_i, y_j} \wedge rec(m_{ij}) \not\leq LS_j^{y_j}$$

That is, channel state $SC_{ik}^{y_i, z_k}$ and process state $LS_k^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.



A global state GS1 consisting of local states $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is **inconsistent** because

- the state of p_2 has recorded the receipt of message m_{12} ,
- however, the state of p_1 has not recorded its send.

On the contrary, a global state GS2 consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent;

- all the channels are empty except C_{21} that contains message m_{21}

Global state contd...

- **Transit-less** : A global state is transit –less, if all channels are recorded empty

A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$ is *transitless* iff

$$\forall i, \forall j : 1 \leq i, j \leq n :: SC_{ij}^{y_i, z_j} = \phi.$$

- **Strongly consistent** : A global state is **Strongly consistent** iff It is transit – less as well as consistent
- **Designing efficient algms for recording global state of a DS is an important Problem**

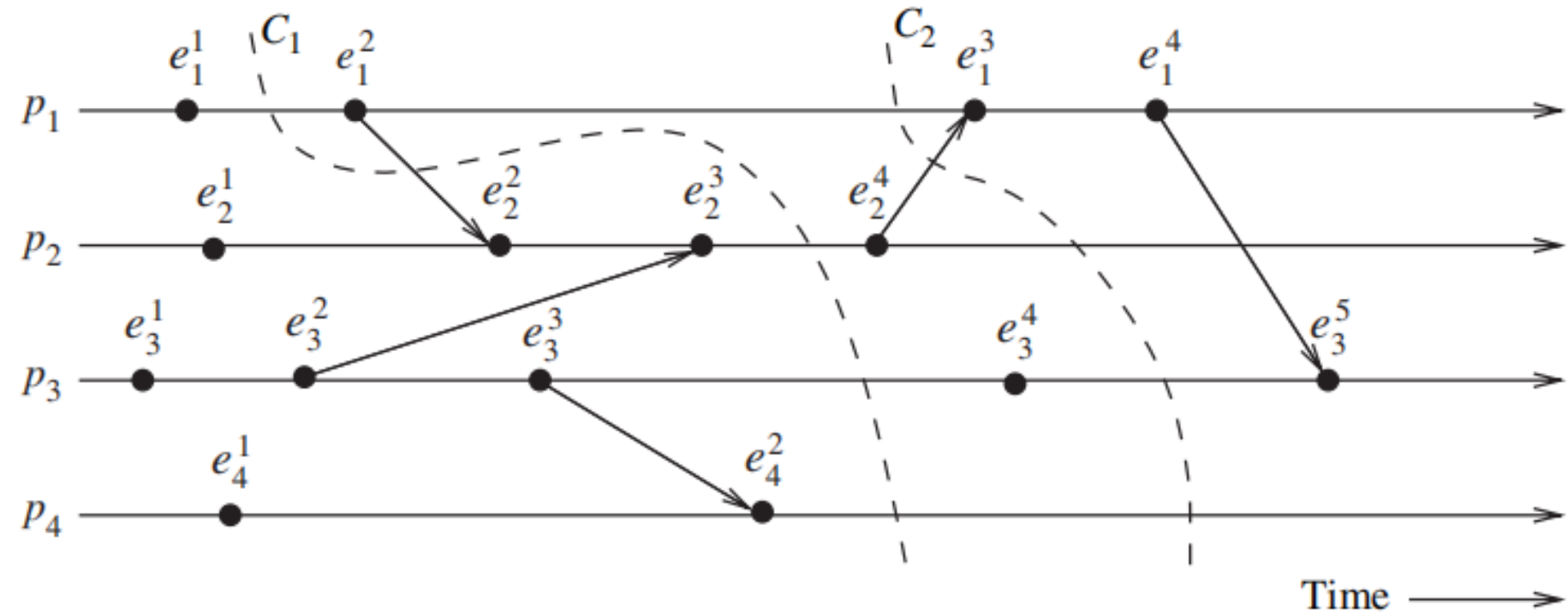
Cuts of a distributed computation

- In the space–time diagram of a distributed computation, a **zigzag line** joining one arbitrary point on each process line is termed **a cut in the computation**.
- **slices the space–time diagram**,
- Sets events in the distributed computation, into a PAST and a FUTURE
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut C , $PAST(C)$ \rightarrow set of events in the PAST, $FUTURE(C)$ denote the FUTURE of C , respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space–time diagram

Cuts of a distributed computation contd...

- ❑ A **consistent global state** corresponds to a **cut** in which every message **received** in the **PAST** of the cut was **sent in the PAST** of that cut.
- ❑ Such a cut is known as a ***consistent cut***.
- ❑ All messages that cross the cut from the PAST to the FUTURE are in **transit** in the corresponding consistent global state.
- ❑ A cut is ***inconsistent*** if a message crosses the cut from the FUTURE to the PAST

Figure 2.3 Illustration of cuts in a distributed execution.



C_1 is an inconsistent cut, whereas C_2 is a consistent cut.

Cuts of a distributed computation contd...

definition 2.1 If $e_i^{Max_PAST_i(C)}$ denotes the latest event at process p_i that is in the **PAST** of a cut C , then the global state represented by the cut is $\{\bigcup_i LS_i^{Max_PAST_i(C)}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$ where $SC_{jk}^{y_j, z_k} = \{m \mid send(m) \in \mathbf{PAST}(C) \wedge rec(m) \in \mathbf{FUTURE}(C)\}$.

Past and future cones of an event

- An event e_j could have been affected only by all events e_i such that $e_i \rightarrow e_j$ and
- all the information available at e_i could be made accessible at e_j
- All such events e_i belong to the past of e_j .
- $Past(e_j)$ denote all events in the past of e_j in a computation (H, \rightarrow) . Then,

$$Past(e_j) = \{e_i | \forall e_i \in H, e_i \rightarrow e_j\}.$$

Let $Past_i(e_j)$ be the set of all those events of $Past(e_j)$ that are on process p_i .

$max(Past_i(e_j))$ is the latest event at process p_i that affected event e_j

Past cone of an event

Let $Past_i(e_j)$ be the set of all those events of $Past(e_j)$ that are on process p_i .

$max(Past_i(e_j))$ is the latest event at process p_i that affected event e_j

$max(Past_i(e_j))$ is always a message send event.

Let $Max_Past(e_j) = \bigcup_{(i)} \{max(Past_i(e_j))\}$. $Max_Past(e_j)$ consists of the latest event at every process that affected event e_j and is referred to as the *surface of the past cone* of e_j

Future cone of an event

The future of an event e_j , $Future(e_j)$, contains all events e_i that are causally affected by e_j . In a computation (H, \rightarrow) , $Future(e_j)$ is defined as:

$$Future(e_j) = \{e_i | \forall e_i \in H, e_j \rightarrow e_i\}.$$

$Future_i(e_j)$ as the set of those events of $Future(e_j)$ that are on process p_i

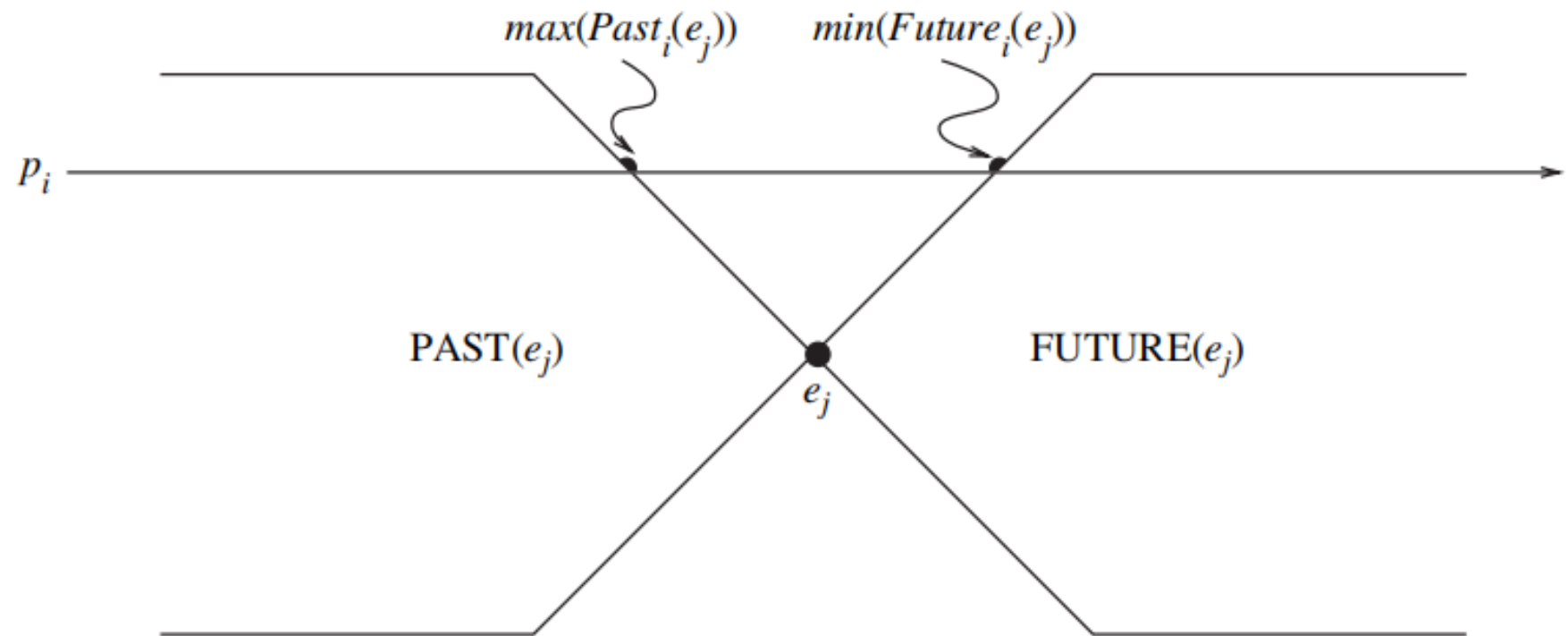
$\min(Future_i(e_j))$ as the first event on process p_i that is affected by e_j .

$\min(Future_i(e_j))$ is always a message receive

Future cone of an event contd...

$Min_Past(e_j)$, defined as $\bigcup_{(v_i)} \{min(Future_i(e_j))\}$, consists of the first event at every process that is causally affected by event e_j and is referred to as the *surface of the future cone* of e_j .

Figure 2.4 Illustration of past and future cones in a distributed computation.



Models of process communications

There are two basic models of process communications

- synchronous
- asynchronous.

Models of process communications contd..

The **synchronous communication** model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process.

- The sender process resumes execution only after it learns that the receiver process has accepted the message.
- Thus, the sender and the receiver processes must synchronize to exchange a message.

Models of process communications

contd..

- **Asynchronous communication** model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
- After having sent a message, the sender process does not wait for the message to be delivered to the receiver process.
- The message is buffered by the system and is delivered to the receiver process when it is ready to accept the message.
- A buffer overflow may occur if a process sends a large a number of messages in a burst to another process.
- Asynchronous communication provides higher parallelism because the sender process can execute while the message is in transit to the receiver
- Due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications