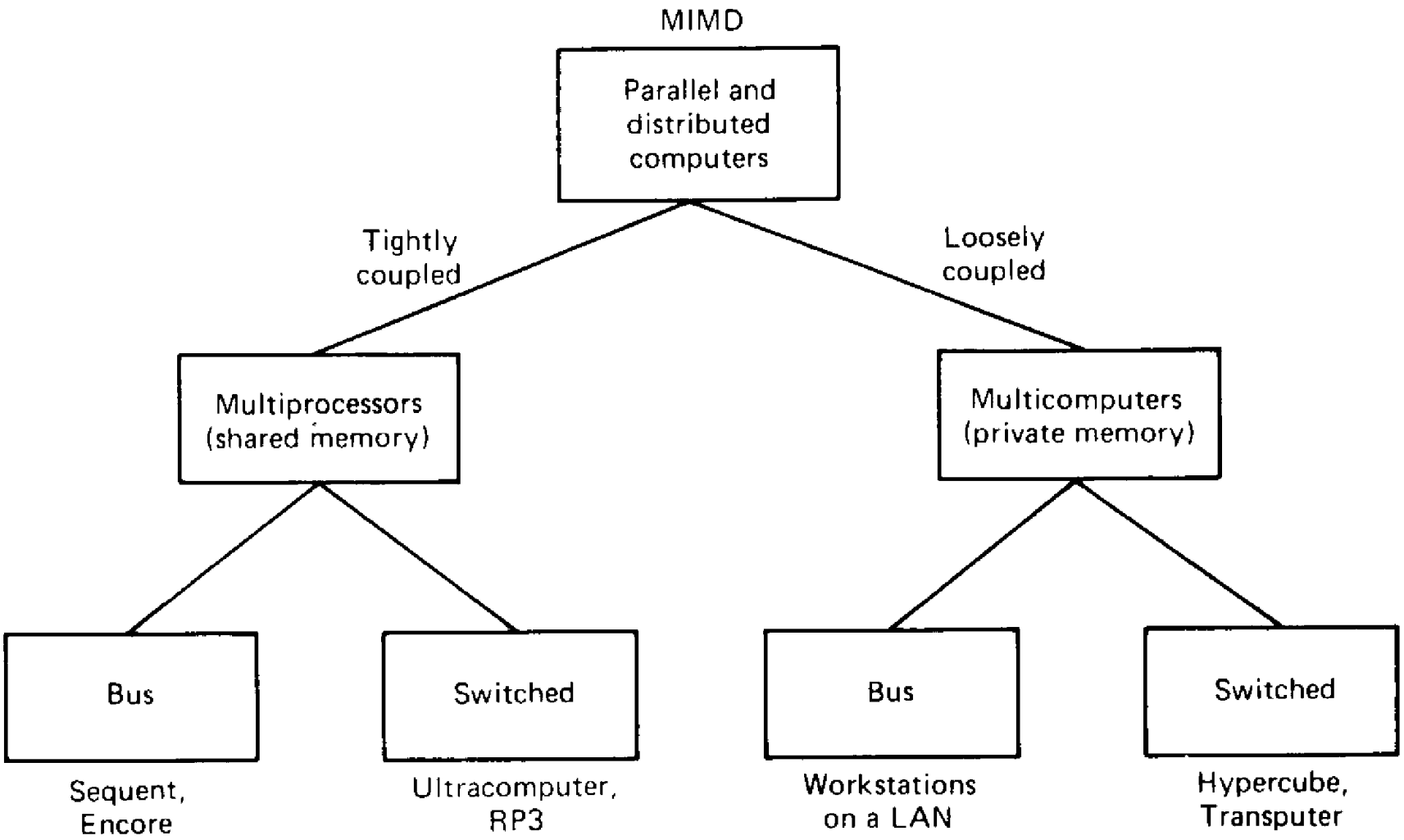
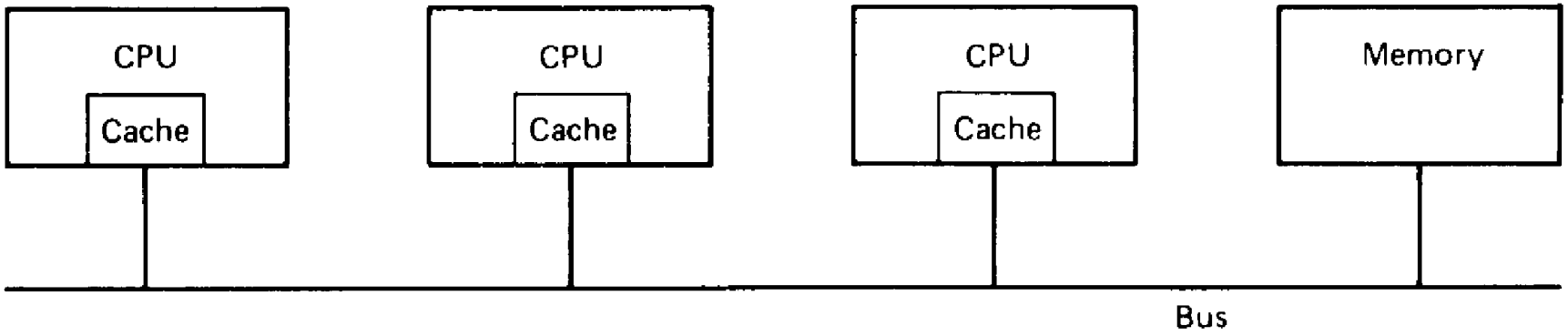
Communication / Interconnection Networks

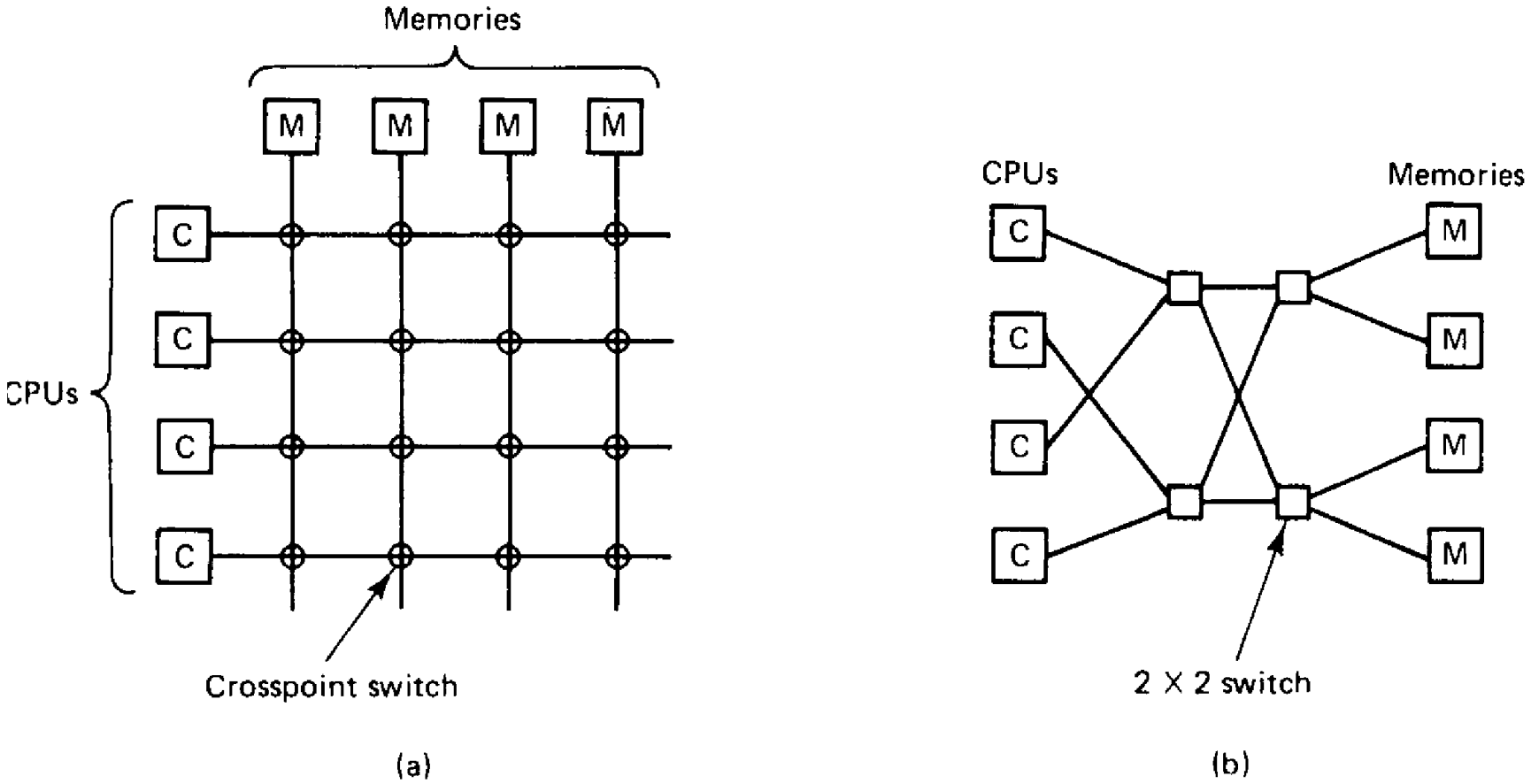

MIMD
Parallel and distributed computers
Tightly coupled
Loosely coupled
Multiprocessors (shared memory)
Multicomputers (private memory)
Bus
Switched
Bus
Switched
Sequent, Encore
Ultracomputer, RP3
Workstations on a LAN
Hypercube, Transputer

Bus based Multiprocessors



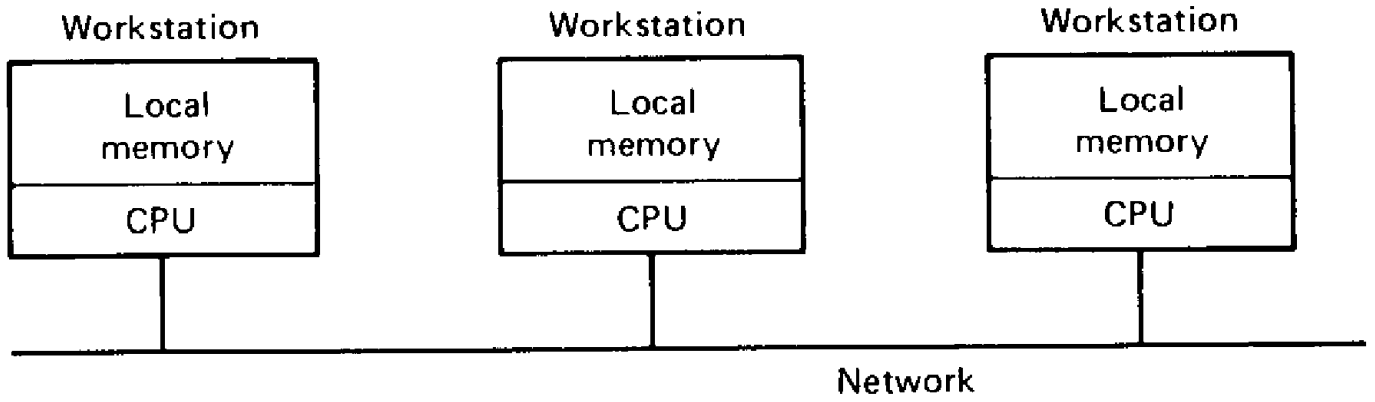
- Data transmitted in the form of packets
- Cache coherency

Switched Multiprocessors



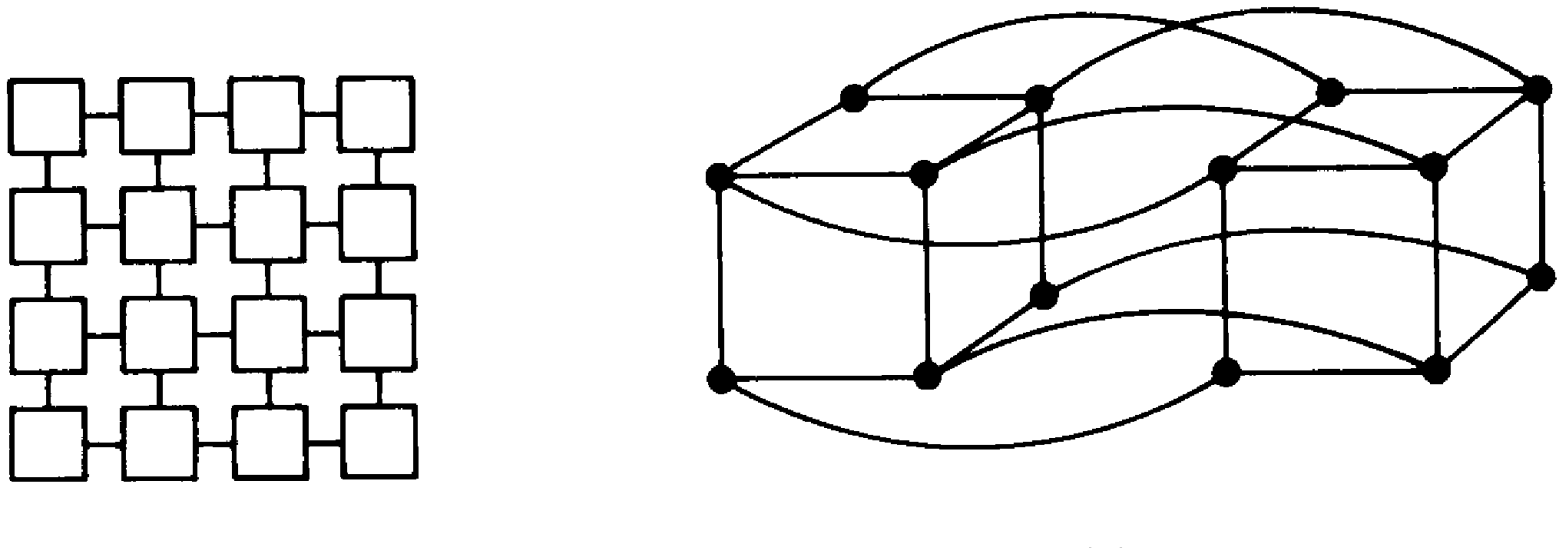
(a) A crossbar switch. (b) An omega switching network.

Bus based Multicomputers



- Limitation – slow as no. of computers increase

Switched Multicomputers



(a) Grid. (b) Hypercube.

Example 1

- A multicomputer with 256 CPUs is organized as a 16 × 16 grid. What is the worst-case delay (in hops) that a message might have to take?
- **A:** Assuming that routing is optimal, the longest optimal route is from one corner of the grid to the opposite corner. The length of this route is 30 hops. If the end processors in a single row or column are connected to each other, the length becomes 15.

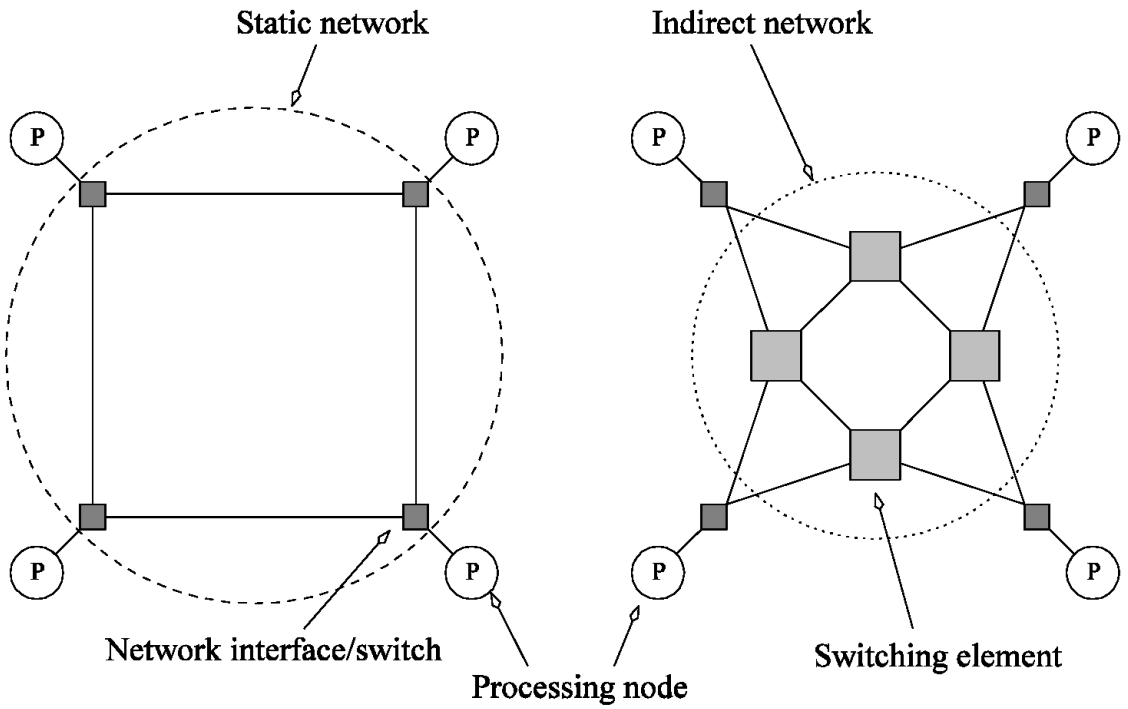
Example 2

- Now consider a 256-CPU hypercube. What is the worst-case delay here, again in hops?
- **A:** With a 256-CPU hypercube, each node has a binary address, from 00000000 to 11111111. A hop from one machine to another always involves changing a single bit in the address. Thus from 00000000 to 00000001 is one hop. From there to 00000011 is another hop. In all, eight hops are needed.

Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as static or dynamic.
- Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

Static and Dynamic Interconnection Networks



Classification of interconnection networks: (a) a static network; and (b) a dynamic network.

Properties of a Topology/Network

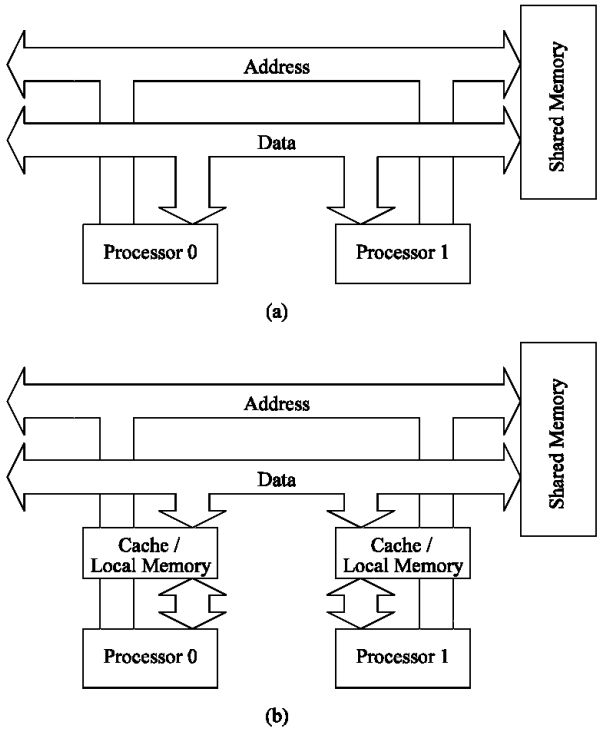
Bisection Bandwidth

- -Often used to describe network performance
- -Cut network in half and sum bandwidth of links severed
- -(Min # channels spanning two halves) * (BW of each channel)
- -Meaningful only for recursive topologies
- -Can be misleading, because does not account for switch and routing efficiency

Many Topology Examples

- Bus
- Crossbar
- Ring
- Tree
- Omega
- Hypercube
- Mesh
- Torus
- Butterfly
- …

Buses

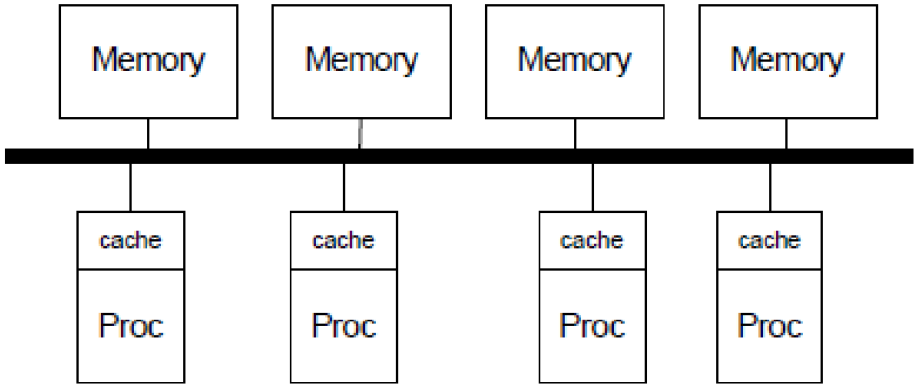


Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.

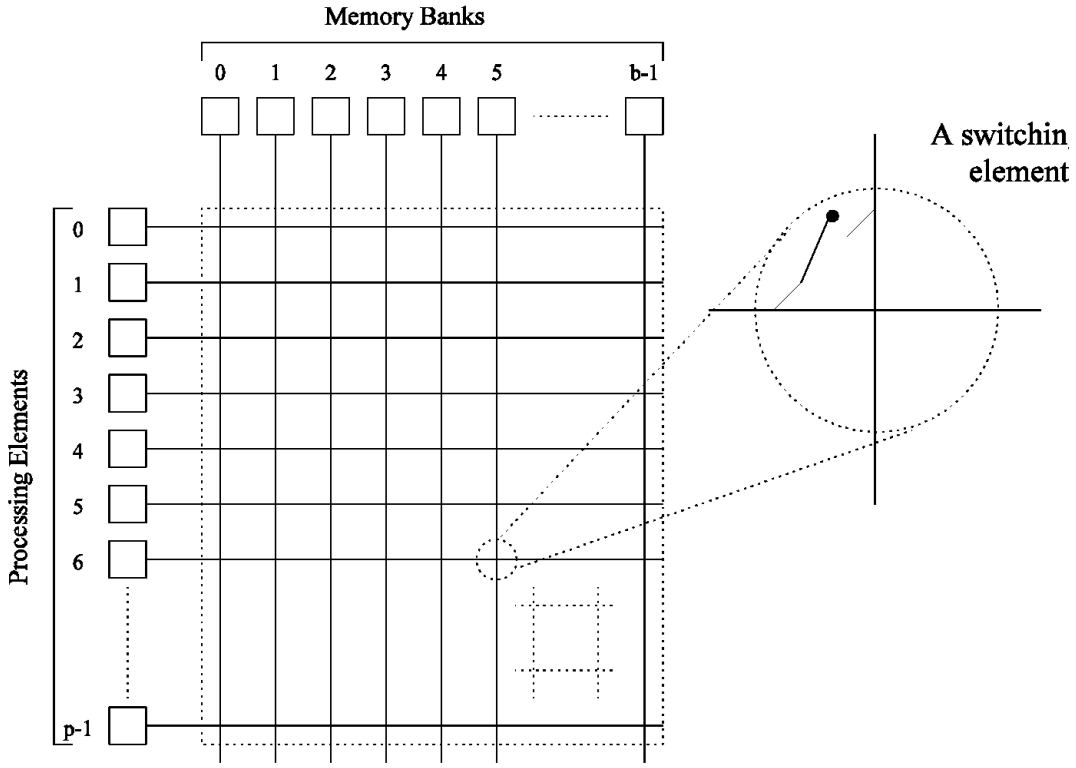
Bus

+ **Simple**
+ **Cost effective for a small number of nodes**
+ **Easy to implement coherence** (snooping)
- **Not scalable to large number of nodes**
  (limited bandwidth, electrical loading → reduced frequency)
- **High contention**

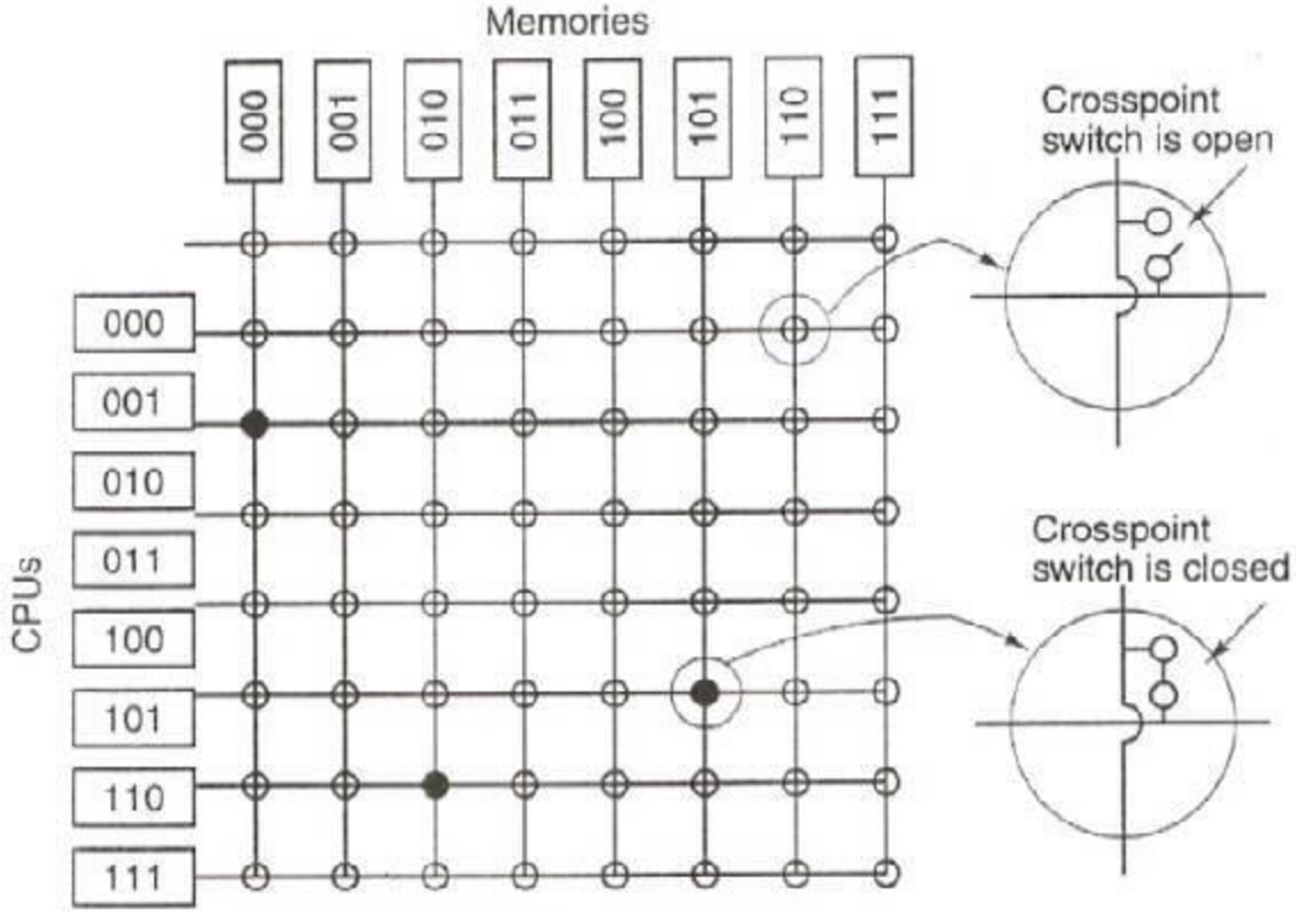


Crossbars

A crossbar network uses an *p*×*m* grid of switches to connect *p* inputs to m outputs in a non-blocking manner.



A completely non-blocking crossbar network connecting *p* processors to b memory banks.


Memories
000
001
010
011
100
101
110
111
CPUs
000
001
010
011
100
101
110
111
Crosspoint switch is open
Crosspoint switch is closed

Crossbar

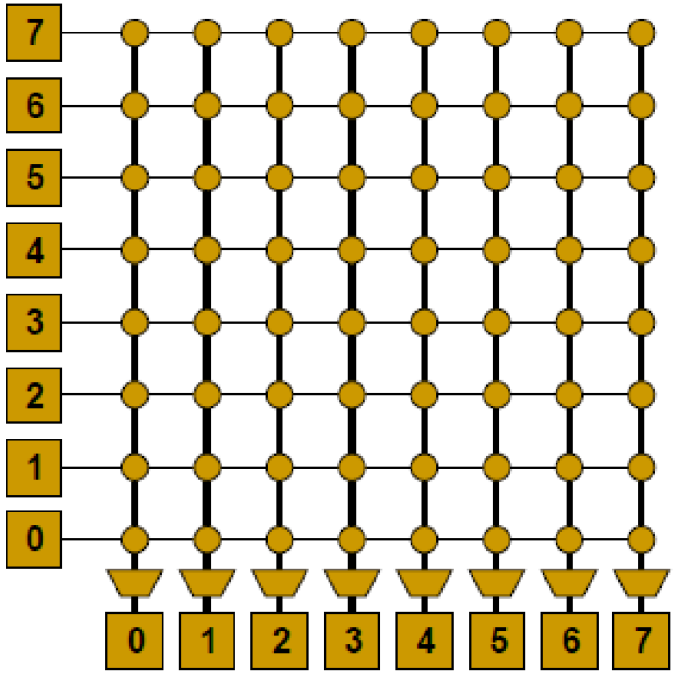
- Every node connected to all others (non-blocking)
- Good for small number of nodes

+ Low latency and high throughput

- Expensive
- Not scalable → O(N^2) cost
- Difficult to arbitrate

Core-to-cache-bank networks:

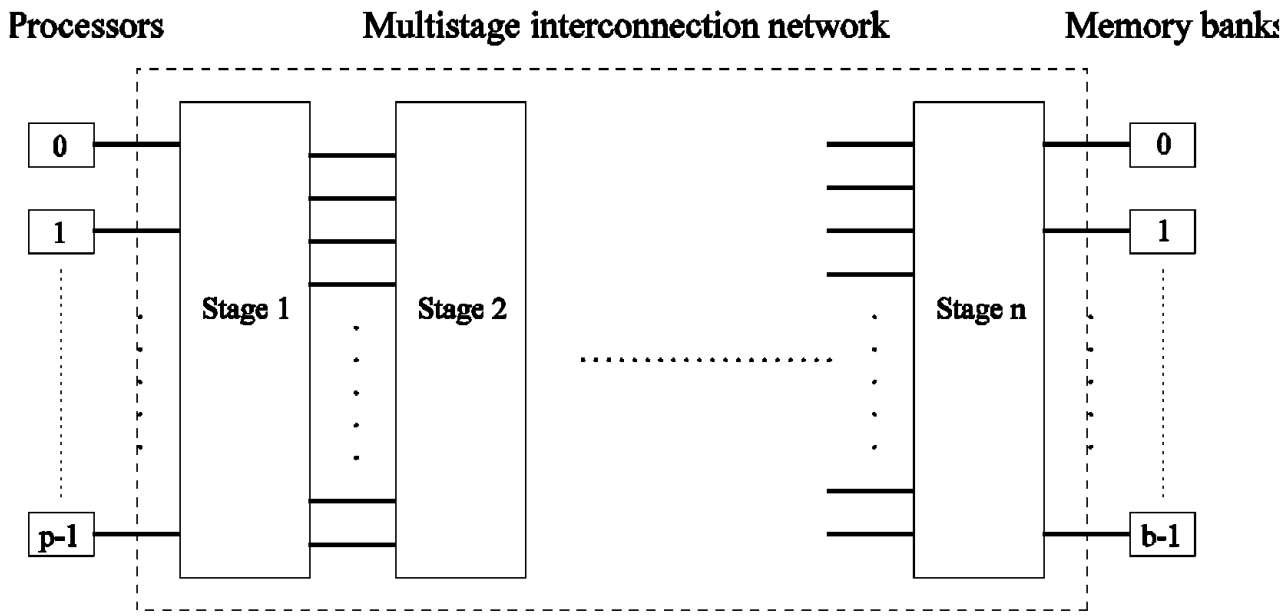
- IBM POWER5
- Sun Niagara I/II



Multistage Networks

- Crossbars have excellent performance scalability but poor cost scalability.
- Buses have excellent cost scalability, but poor performance scalability.
- Multistage interconnects strike a compromise between these extremes.

Multistage Networks



The schematic of a typical multistage interconnection network.

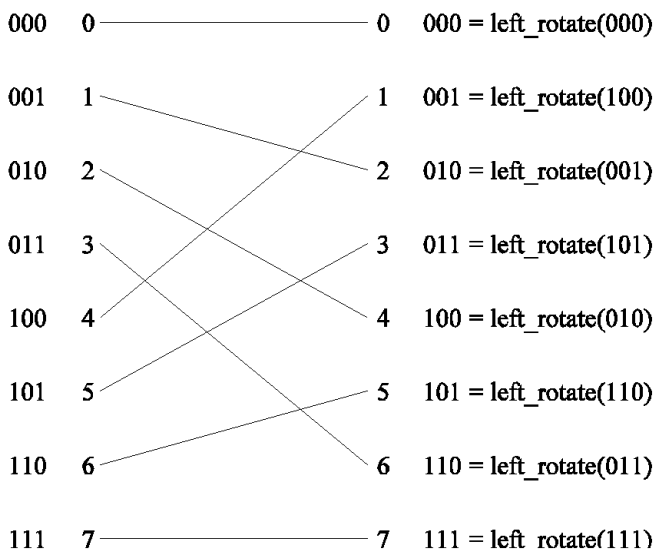
Multistage Networks

- One of the most commonly used multistage interconnects is the Omega network.
- This network consists of *log p* stages, where *p* is the number of inputs/outputs.
- At each stage, input *i* is connected to output *j* if:

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases}$$

Multistage Omega Networks

Each stage of the Omega network implements a perfect shuffle as follows:



A perfect shuffle interconnection for eight inputs and outputs.

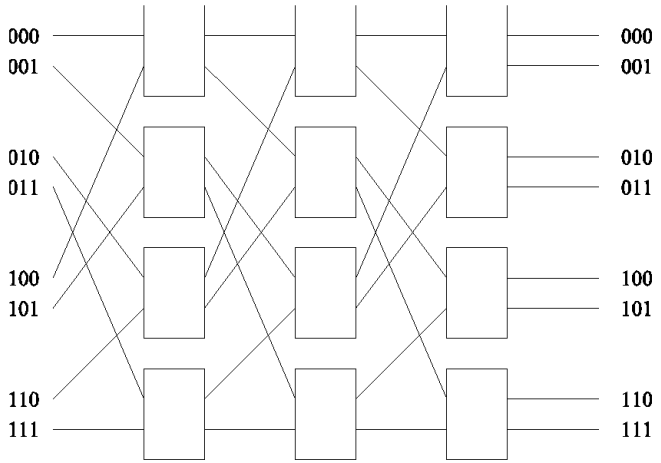
Multistage Omega Network

- The perfect shuffle patterns are connected using 2×2 switches.
- The switches operate in two modes – crossover or passthrough.

Two switching configurations of the 2 × 2 switch:
(a) Pass-through; (b) Cross-over.

Multistage Omega Network

A complete Omega network with the perfect shuffle interconnects and switches can now be illustrated:



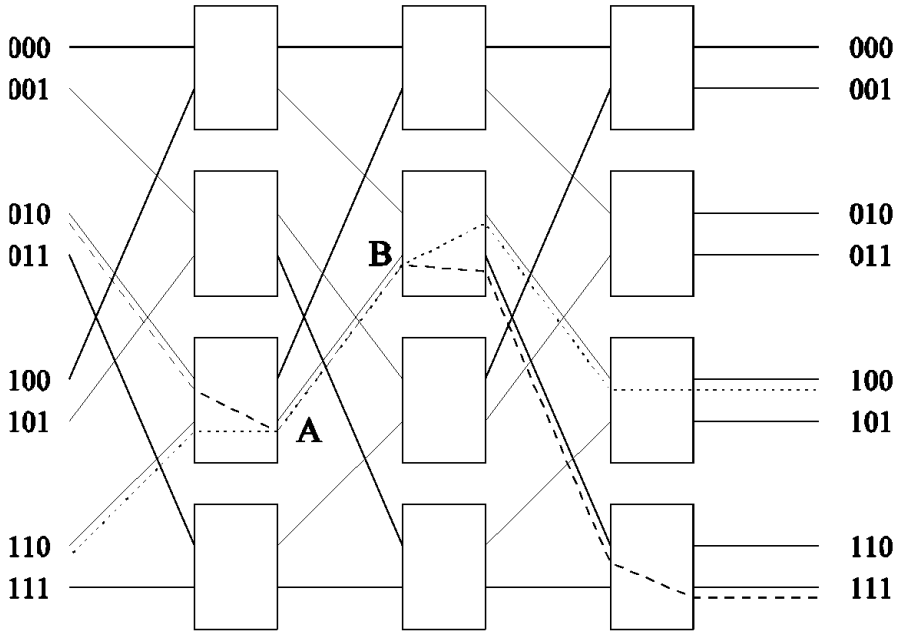
A complete omega network connecting eight inputs and eight outputs.

An omega network has $p/2 \times \log p$ switching nodes, and the cost of such a network grows as $(p \log p)$.

Multistage Omega Network – Routing

- Let *s* be the binary representation of the source and *d* be that of the destination processor.
- The data traverses the link to the first switching node. If the most significant bits of *s* and *d* are the same, then the data is routed in pass-through mode by the switch else, it switches to crossover.
- This process is repeated for each of the *log p* switching stages.
- Note that this is not a non-blocking switch.

Multistage Omega Network – Routing



An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

Example 3

- A multiprocessor has 1024 100-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be to allow a request to go to memory and back in one instruction time?

- Solution: 5120 0.5 nano sec switches

Example 4

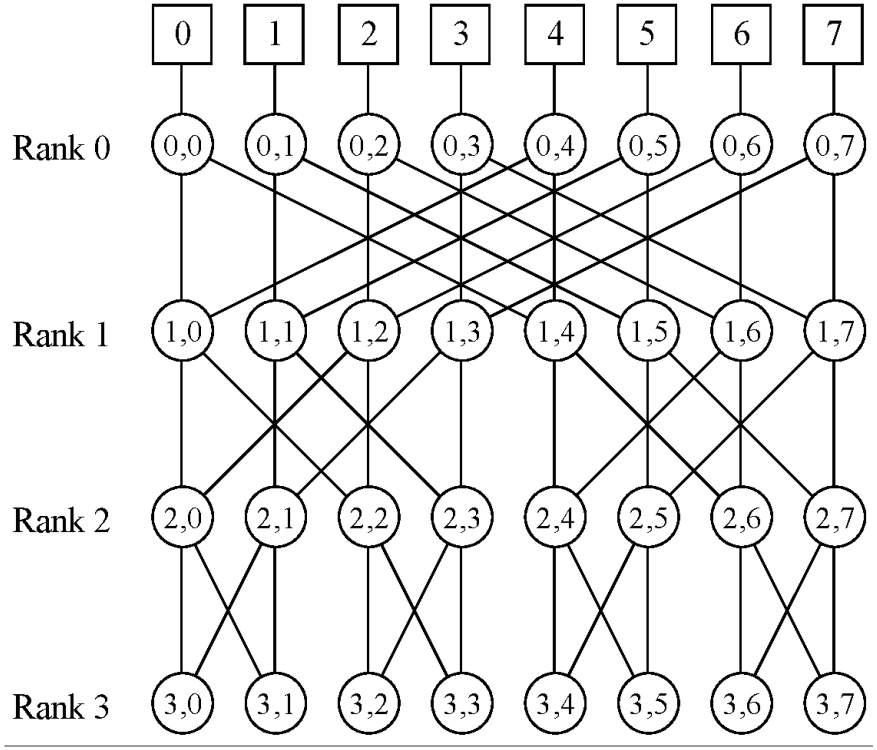
- A multiprocessor has 4096 50-MIPS CPUs connected to memory by an omega network. How fast do the switches have to be to allow a request to go to memory and back in one instruction time?
- Solution: 1 instruction= 20 nano sec

  1 switching stage=0.833 nano sec

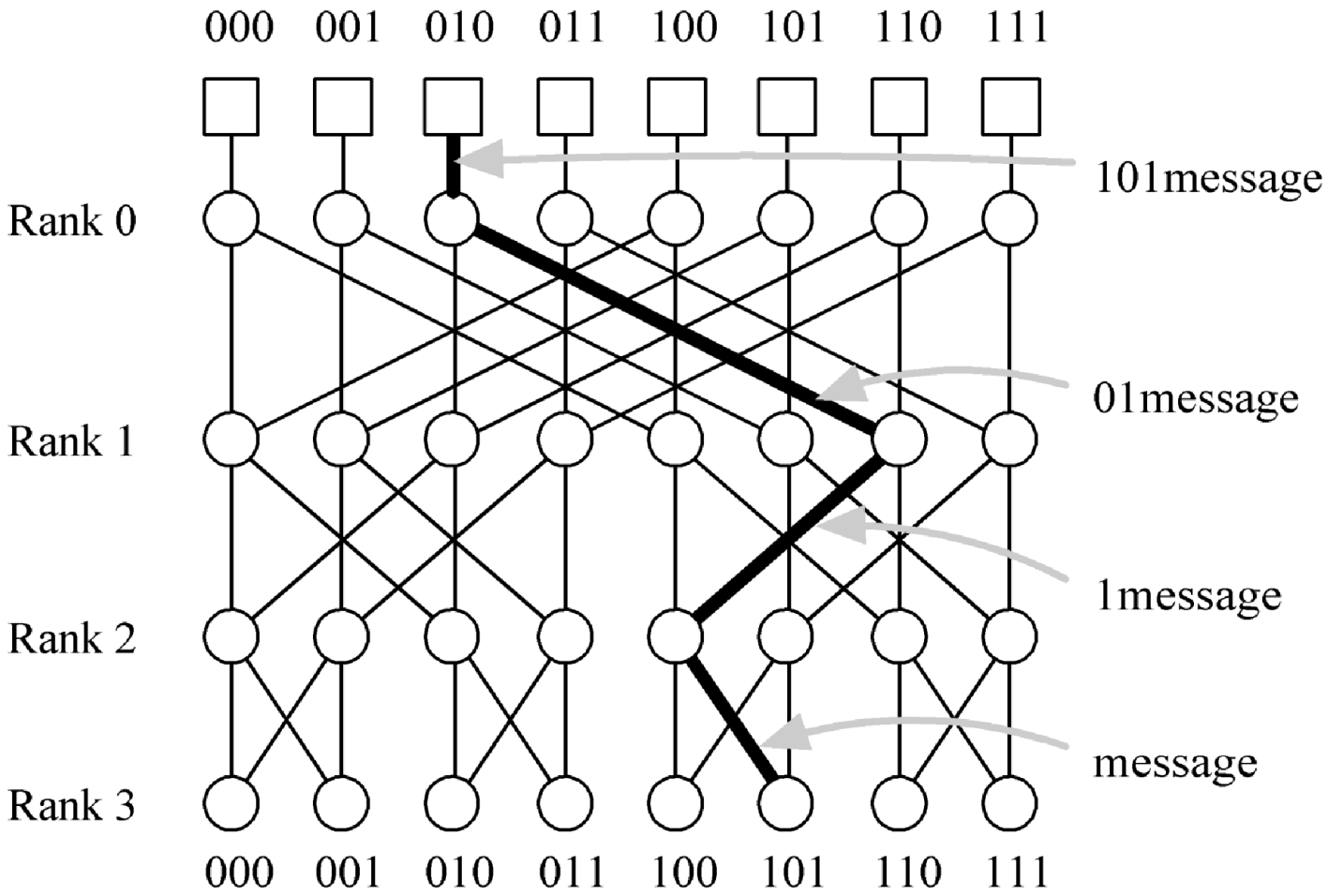
  2048x12 0.833 nano second switches

Butterfly Network

- Indirect topology
- $n = 2^d$ processor nodes connected by $n(\log n + 1)$ switching nodes
- Rows are labeled 0 … n. Each processor has four connections to other processors (except processors in top and bottom row).
- Processor P(r, j), i.e. processor number j in row r is connected to P(r-1, j) and P(r-1, m) where m is obtained by inverting the r^{th} significant bit in the binary representation of j.



Butterfly Network Routing



Evaluating Butterfly Network

- Diameter: log *n*
- Bisection width: *n* / 2
- Edges per node: 4
- Constant edge length? No

Completely Connected Network

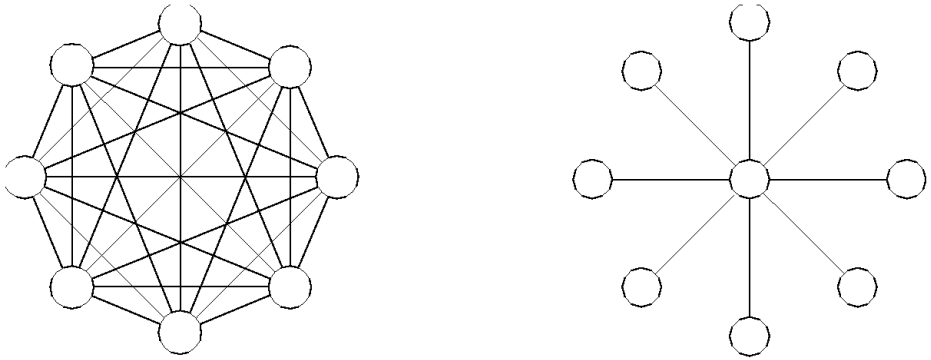
- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of p.
- In this sense, these networks are static counterparts of crossbars.

Star Connected Network

- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is *O(1).* However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.

Completely Connected and Star Connected Networks

Example of an 8-node completely connected network.

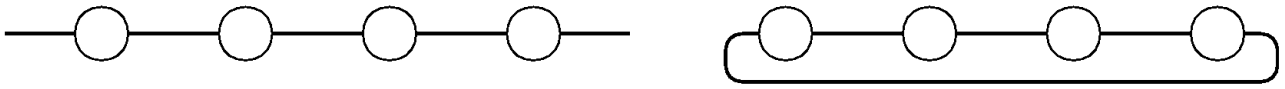


(a) A completely-connected network of eight nodes;
(b) a star connected network of nine nodes.

Linear Arrays, Meshes, and *k-d* Meshes

- In a linear array, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a 1-D torus or a ring.
- A generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.
- A further generalization to *d* dimensions has nodes with *2d* neighbors.
- A special case of a *d*-dimensional mesh is a hypercube. Here, $d = \log p$, where p is the total number of nodes.

Linear Arrays



(a) (b)

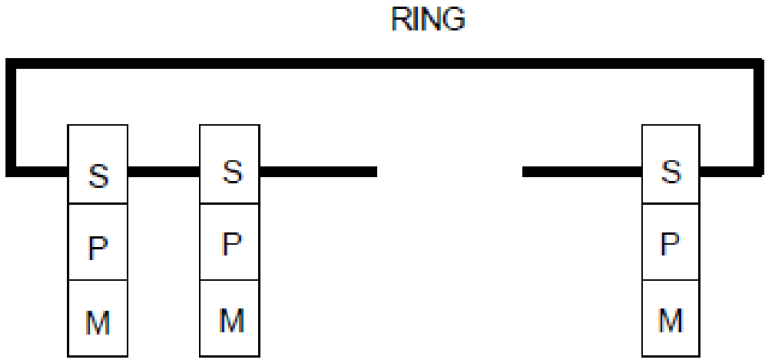
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

Ring

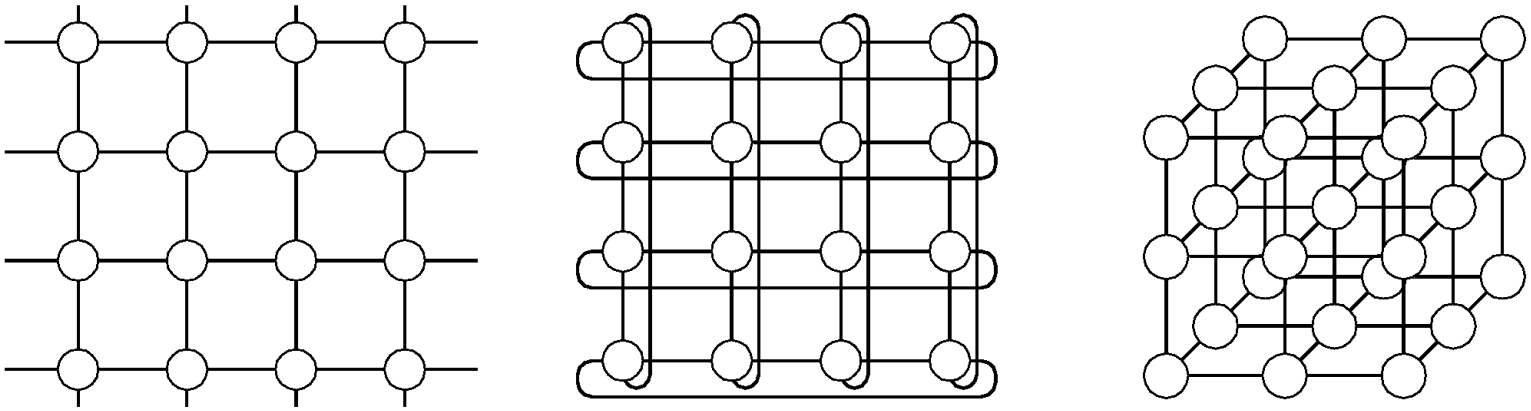
+ Cheap: O(N) cost
- High latency: O(N)
- Not easy to scale
  - Bisection bandwidth remains constant

Used in:

- Intel Larrabee/Core i7
- IBM Cell



Two- and Three Dimensional Meshes

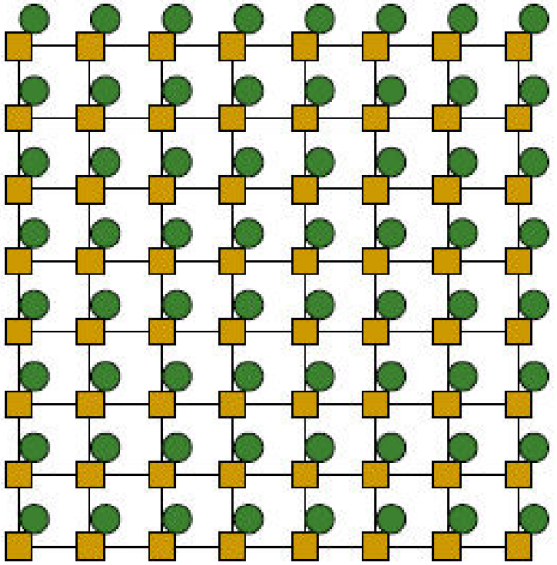


Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and (c) a 3-D mesh with no wraparound.

Mesh

- O(N) cost
- Average latency: O(sqrt(N))
- Easy to layout on-chip: regular & equal-length links
- Path diversity: many ways to get from one node to another

- Used in:
  - Tilera 100-core CMP
  - On-chip network prototypes



Torus

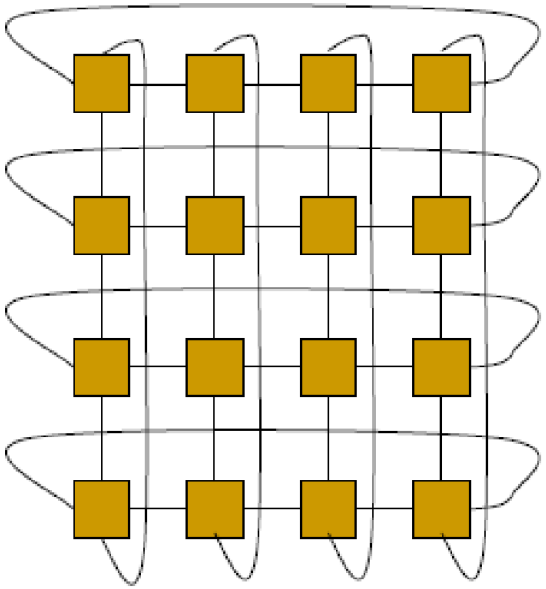
- Mesh is not symmetric on edges: performance very sensitive to placement of task on edge vs. middle
- Torus avoids this problem

+ Higher path diversity (& bisection bandwidth) than mesh

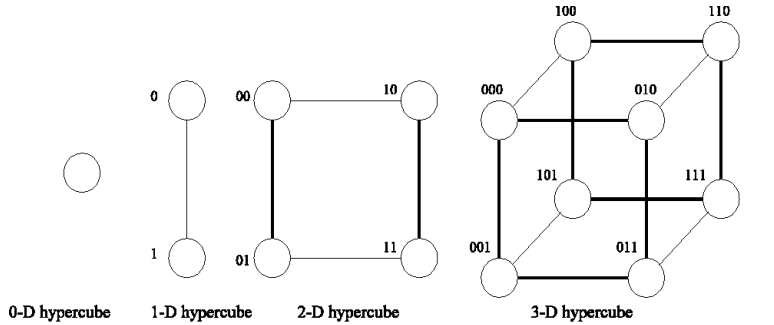
- Higher cost

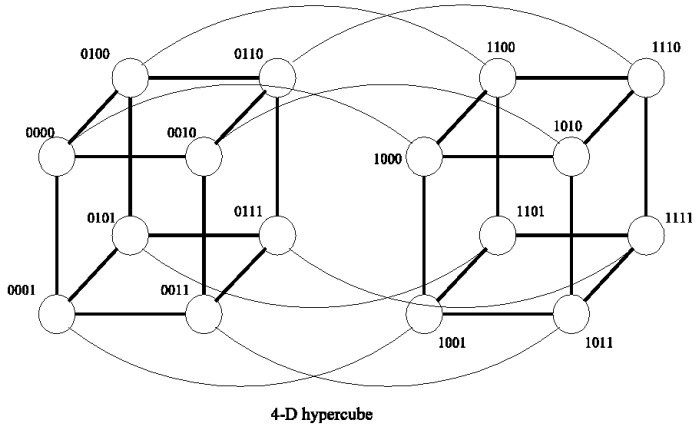
- Harder to lay out on-chip

  - Unequal link lengths



Hypercubes and their Construction





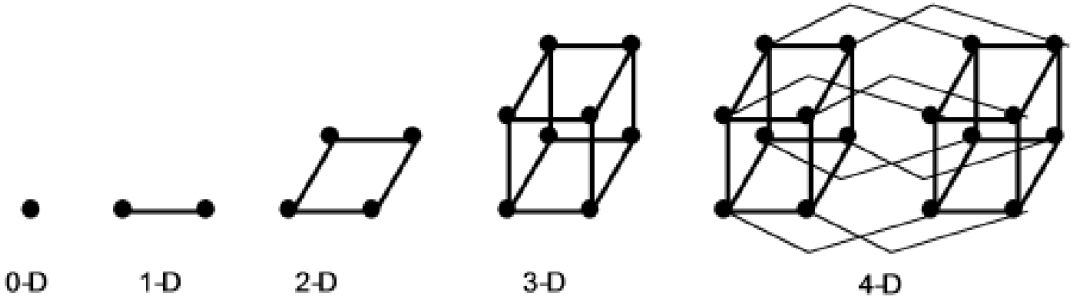
Construction of hypercubes from hypercubes of lower dimension.

Properties of Hypercubes

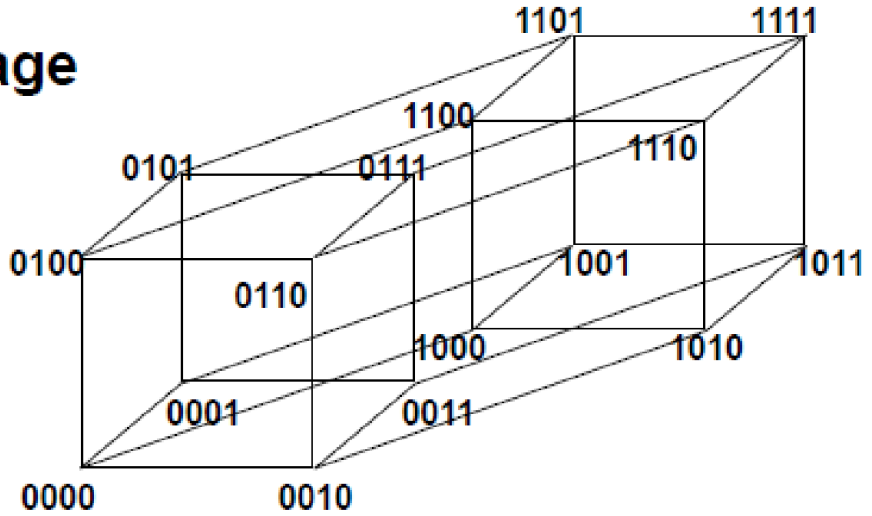
- The distance between any two nodes is at most *log p*.
- Each node has *log p* neighbors.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.

Hypercube

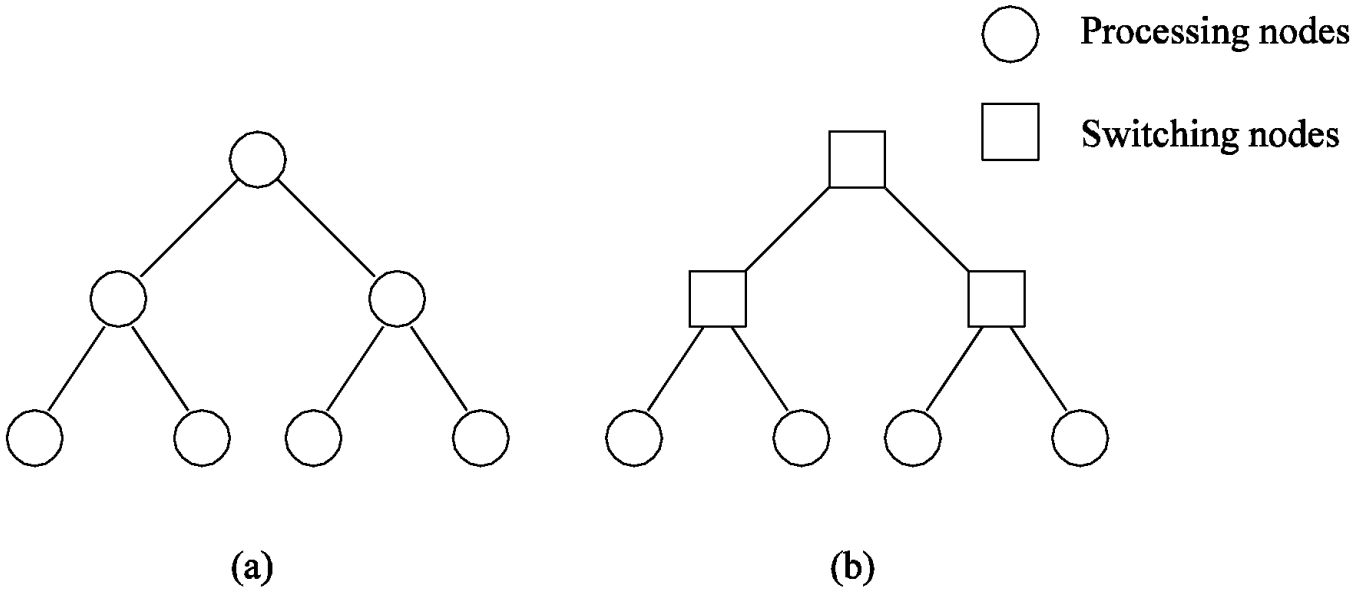
- Latency: O(logN)
- Radix: O(logN)
- #links: O(NlogN)



+ Low latency
- Hard to lay out in 2D/3D
- Used in some early message passing machines, e.g.:
  - Intel iPSC
  - nCube



Tree-Based Networks

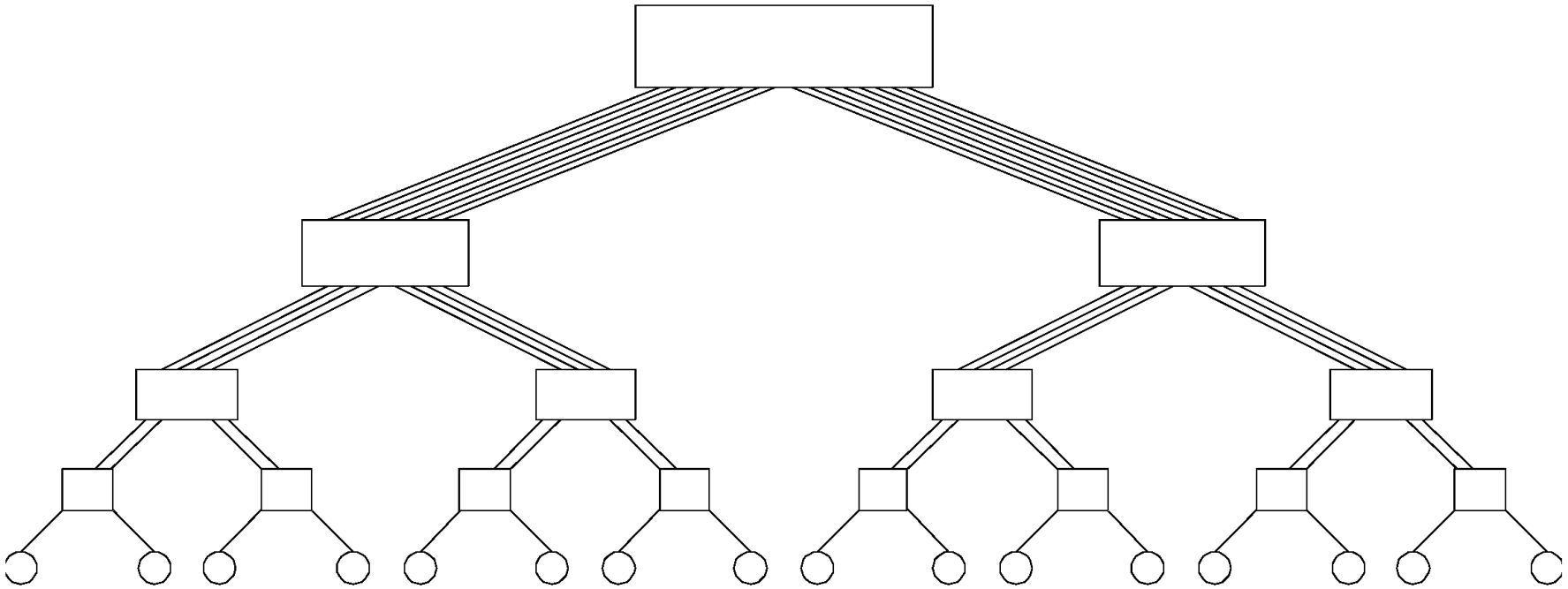


Complete binary tree networks: (a) a static tree network; and (b) a dynamic tree network.

Tree Properties

- The distance between any two nodes is no more than *2logp*.
- Links higher up the tree potentially carry more traffic than those at the lower levels.
- For this reason, a variant called a fat-tree, fattens the links as we go up the tree.
- Trees can be laid out in 2D with no wire crossings. This is an attractive property of trees.

Fat Trees



A fat tree network of 16 processing nodes.

Trees

Planar, hierarchical topology

Latency: O(logN)

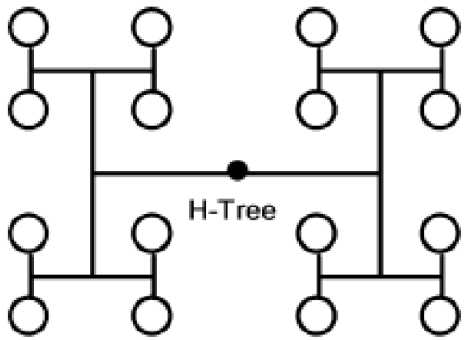
Good for local traffic

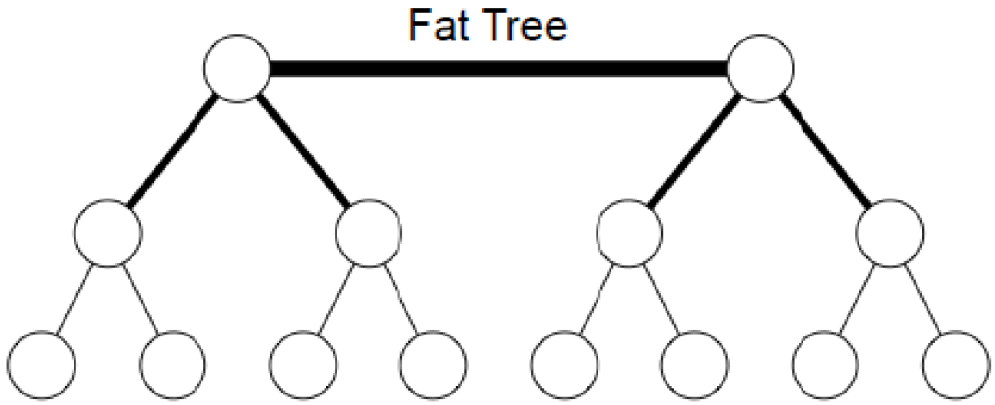
+ Cheap: O(N) cost

+ Easy to Layout

- Root can become a bottleneck

  Fat trees avoid this problem (CM-5)





Evaluating Static Interconnection Networks

- *Diameter:* The distance between the farthest two nodes in the network. The diameter of a linear array is $p - 1$, that of a mesh is $2(\sqrt{p} - 1)$, that of a tree and hypercube is $\log p$, and that of a completely connected network is $O(1)$.
- *Bisection Width:* The minimum number of wires you must cut to divide the network into two equal parts. The bisection width of a linear array and tree is 1, that of a mesh is $\sqrt{p}$, that of a hypercube is $p/2$ and that of a completely connected network is $p^2/4$.
- *Cost:* The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to layout the network, the length of wires, etc., also factor in to the cost.

Evaluating Static Interconnection Networks

- The number of bits that can be communicated simultaneously over a link connecting two nodes is called the ***channel width.*** Channel width is equal to the number of physical wires in each communication link.
- The peak rate at which a single physical wire can deliver bits is called the ***channel rate.***
- The peak rate at which data can be communicated between the ends of a communication link is called ***channel bandwidth.***
- Channel bandwidth is the product of channel rate and channel width.
- The ***bisection bandwidth*** of a network is defined as the minimum volume of communication allowed between any two halves of the network. It is the product of the bisection width and the channel bandwidth. Bisection bandwidth of a network is also sometimes referred to as ***crosssection bandwidth.***

Evaluating Static Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound *k*-ary *d*-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

Evaluating Dynamic Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2 \log p$	1	2	$p - 1$

Communication Primitives

Process Communication

- The single most important difference between a distributed system and a uniprocessor system is the **interprocess communication.**

- In a uniprocessor system, interprocess communication assumes the existence of shared memory.
- A typical example is the producer-consumer problem.

- One process **writes to → buffer →reads from** another process
- The most basic form of synchronization, the semaphore requires **one word** (the semaphore variable) to be shared.

Process Communication

- In a distributed system, there's no shared memory, so the entire nature of interprocess communication must be completely rethought from scratch.
- All communication in distributed system is based on **message passing**.

  E.g. Proc. A wants to communicate with Proc. B

  1. It first builds a message in its own address space
  2. It executes a system call
  3. The OS fetches the message and sends it through network to B.

Process Communication

- A and B have to agree on the meaning of the bits being sent. For example,

- How many volts should be used to signal a 0-bit? 1-bit?
- How does the receiver know which is the last bit of the message?
- How can it detect if a message has been damaged or lost?
- What should it do if it finds out?
- How long are numbers, strings, and other data items? And how are they represented?

Types of Communication Primitives

Widely used communication primitives in Distributed Operating Systems

- Message Passing
  - Using Send() and Receive() Primitives
- Remote Procedure Calls
  - An extension of conventional procedure call (used for transfer of control and data within a single process)

Message Passing Model

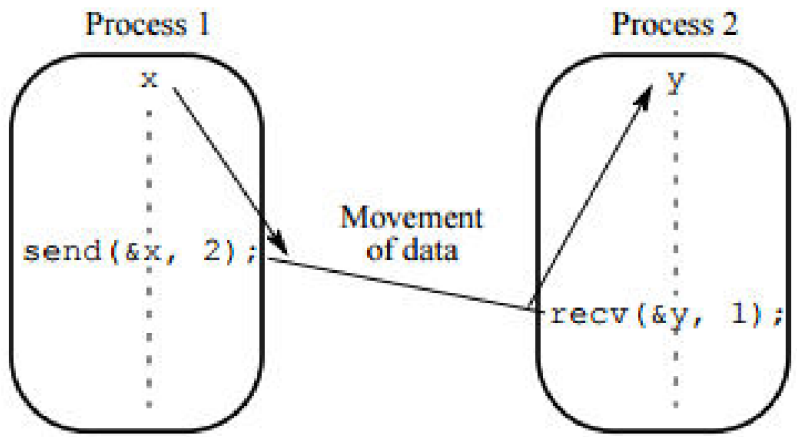
- Two basic communication primitives
  - Send() and Receive()
  - Example:

    send(&x, destination_id)

    receive(&y, source_id)
- Found in Client/Server computing models

Message passing

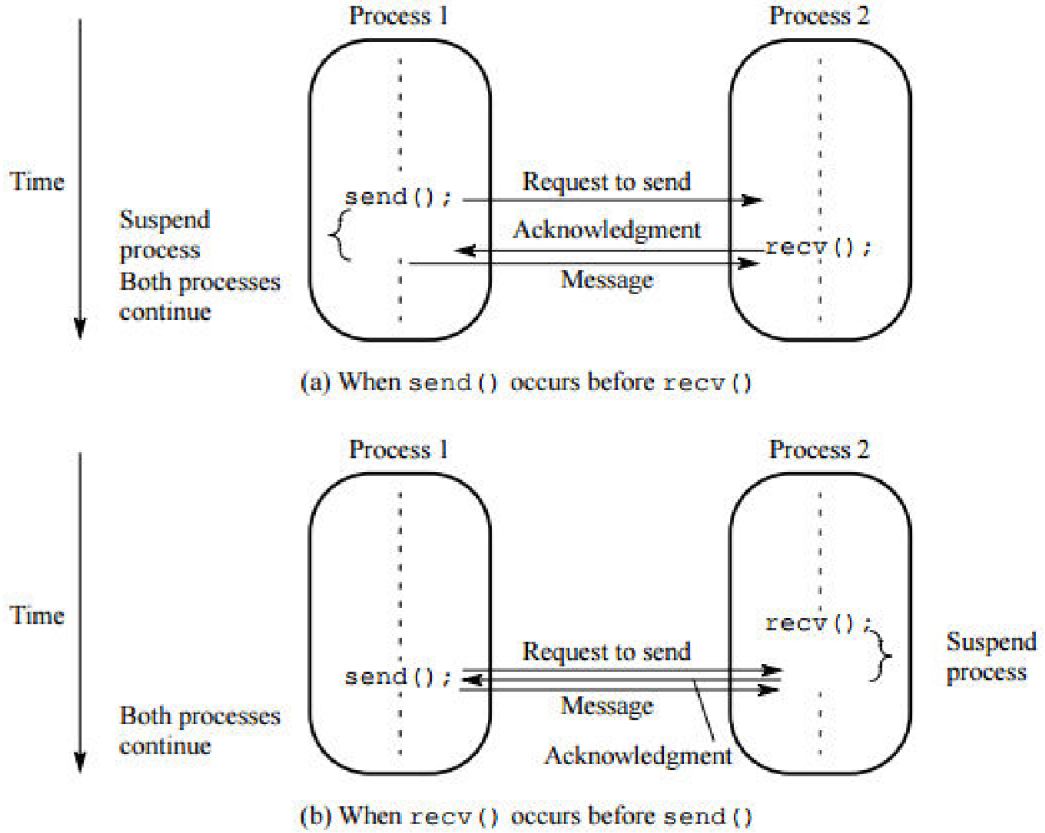
- Synchronous and asynchronous send routines

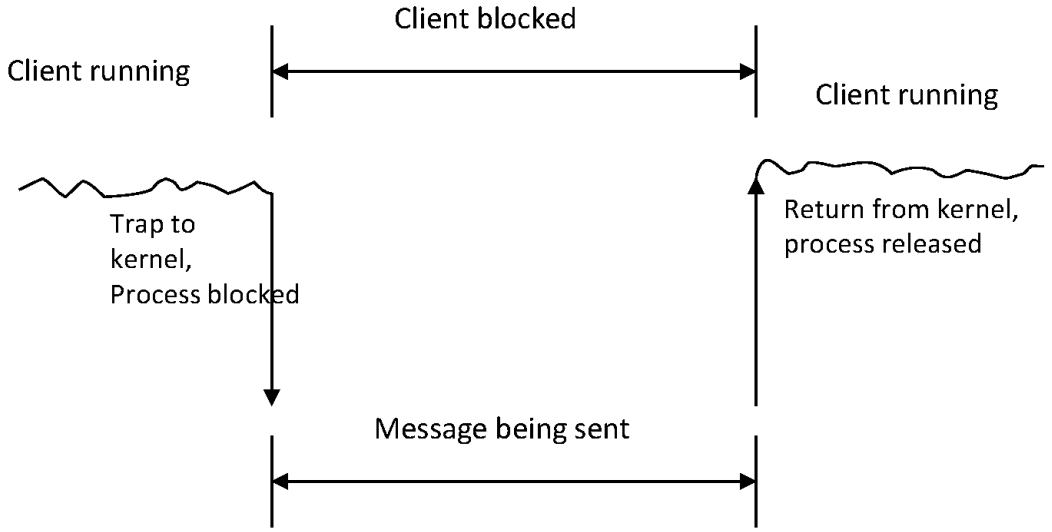


Passing a message between processes using `send()` and `recv()` library calls.

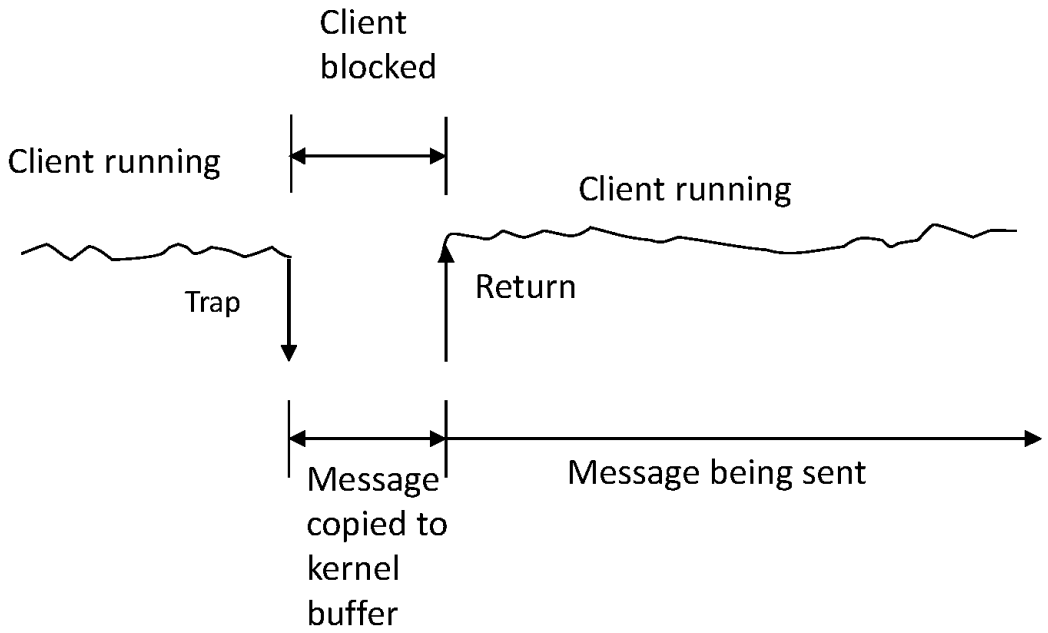
Message passing

- Blocking and non-blocking routines



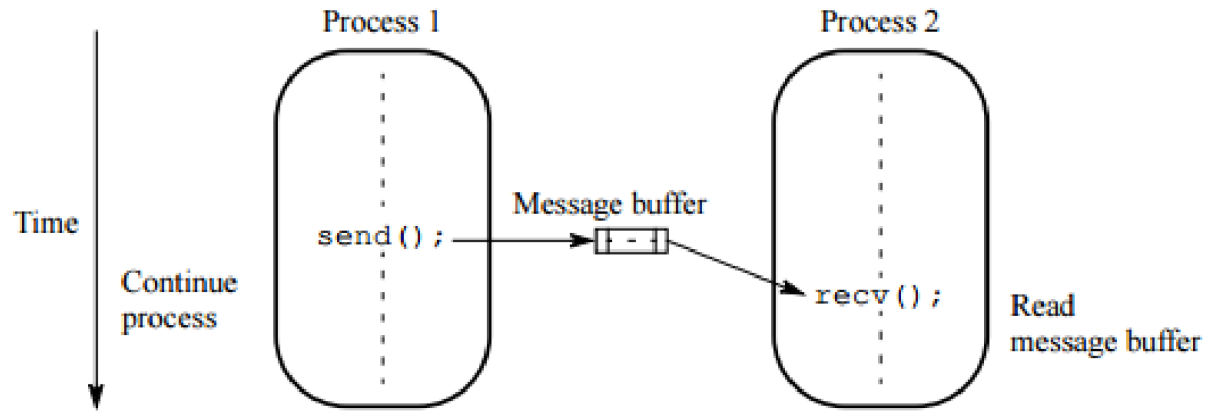


Blocking send primitive



Non-Blocking send primitive

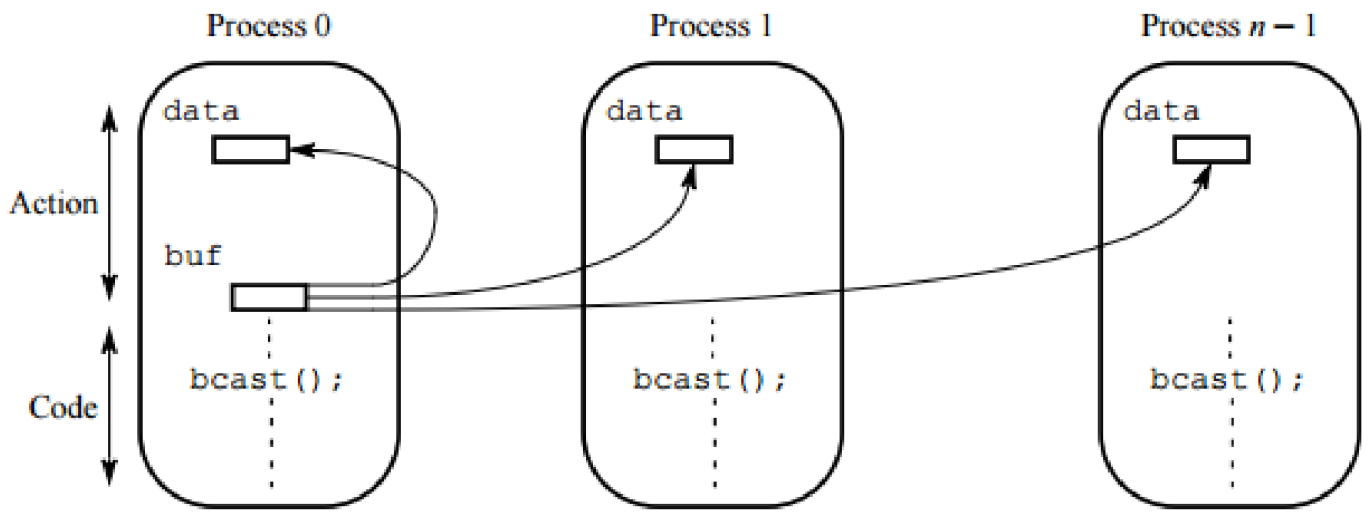
Using a message buffer



Buffered versus Unbuffered Primitives

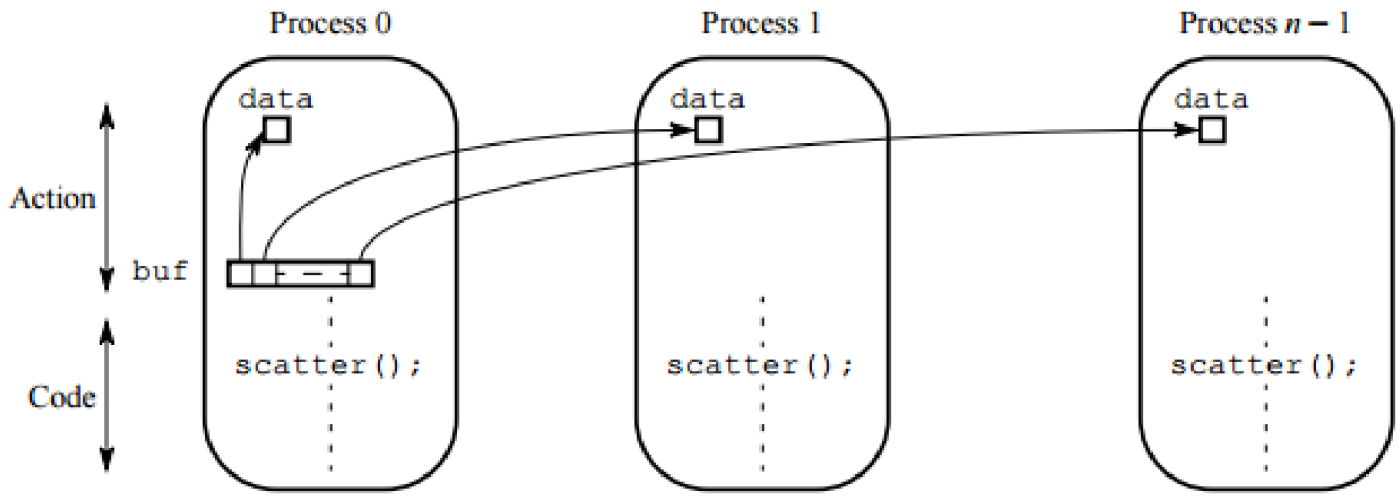
- No buffer allocated. Fine if receive() is called before send().
- Buffers allocated, freed, and managed to store the incoming message. Usually a mailbox created.

Group communication routines



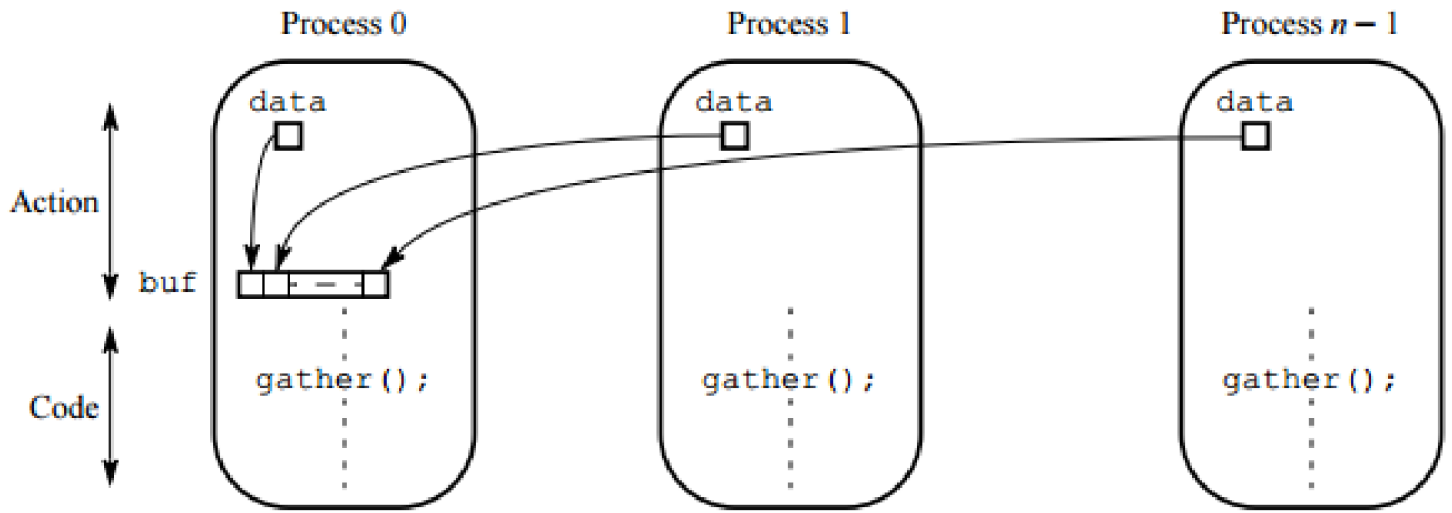
Broadcast operation.

Group communication routines



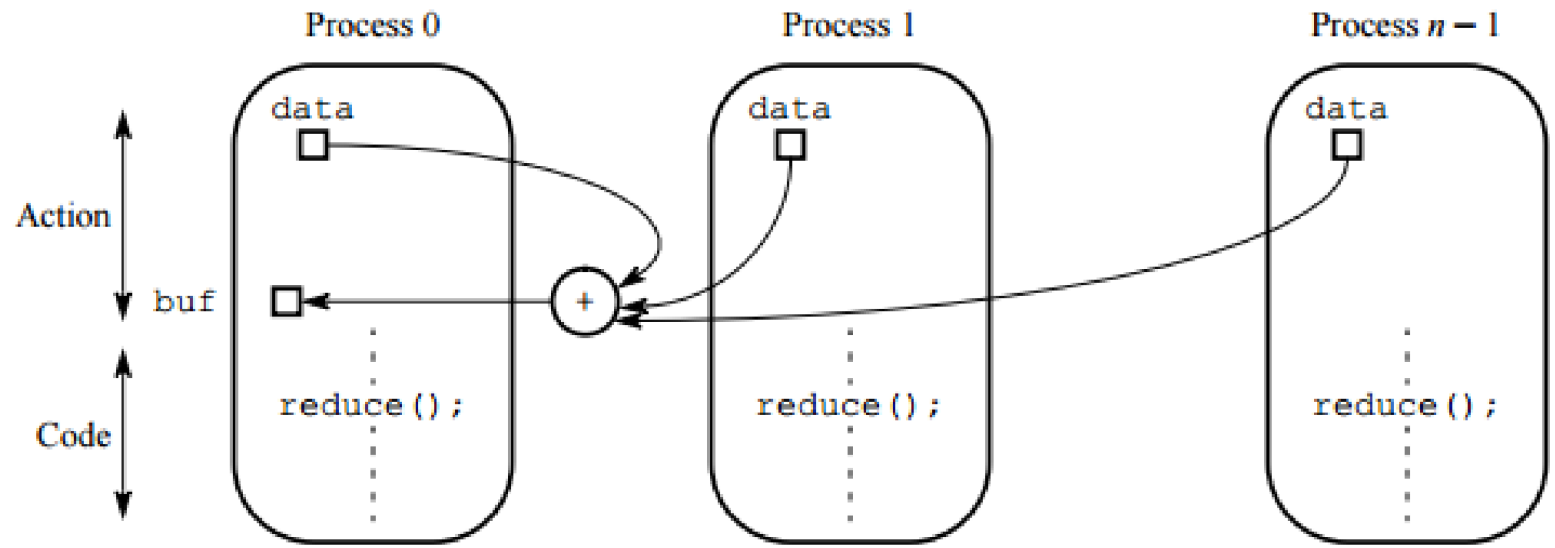
Scatter operation.

Group communication routines



Gather operation.

Group communication routines



Reduce operation (addition).

Example

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000

void main(int argc, char *argv)
{
   int myid, numprocs;
   int data[MAXSIZE], i, x, low, high, myresult, result;
   char fn[255];
   char *fp;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   if (myid == 0) {            /* Open input file and initialize data */
      strcpy(fn,getenv("HOME"));
      strcat(fn,"/MPI/rand_data.txt");
      if ((fp = fopen(fn,"r")) == NULL) {
         printf("Can't open the input file: %s\n\n", fn);
         exit(1);
      }
      for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
   }

   /* broadcast data */
   MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);
```

Example

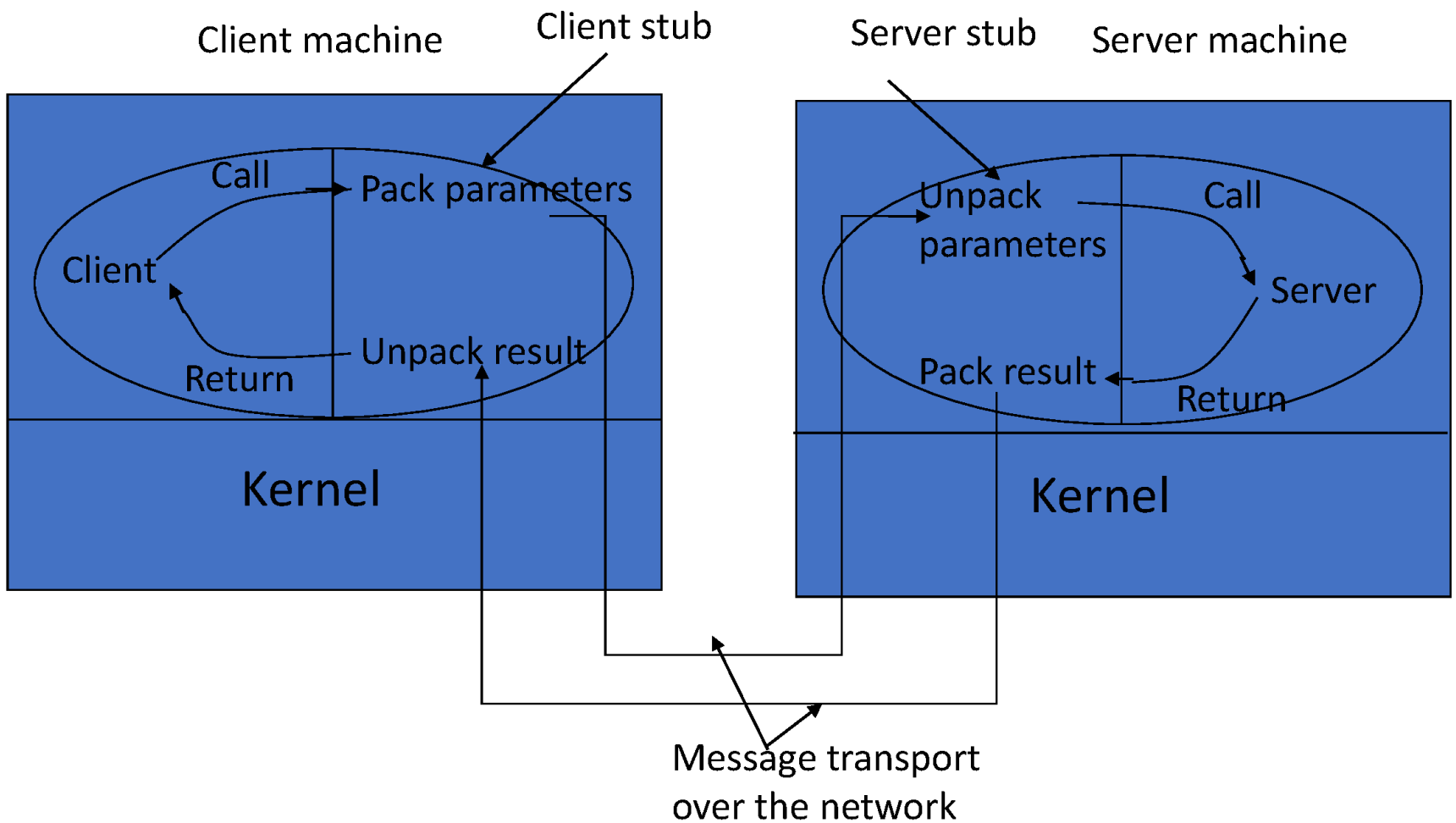
```
/* Add my portion Of data */
   x = n/nproc;
   low = myid * x;
   high = low + x;
   for(i = low; i < high; i++)
      myresult += data[i];
   printf("I got %d from %d\n", myresult, myid);

/* Compute global sum */
   MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (myid == 0) printf("The sum is %d.\n", result);

   MPI_Finalize();
}
```

Remote Procedure Call

- The idea behind RPC is to make a remote procedure call look as much as possible like a local one.
- A remote procedure call occurs in the following steps:

  - The client procedure calls the client stub in the normal way.
  - The client stub builds a message and traps to the kernel.
  - The kernel sends the message to the remote kernel.
  - The remote kernel gives the message to the server stub.
  - The server stub unpacks the parameters and calls the server.
  - The server does the work and returns the result to the stub.
  - The server stub packs it in a message and traps to the kernel.
  - The remote kernel sends the message to the client's kernel.
  - The client's kernel gives the message to the client stub.
  - The stub unpacks the result and returns to the client.

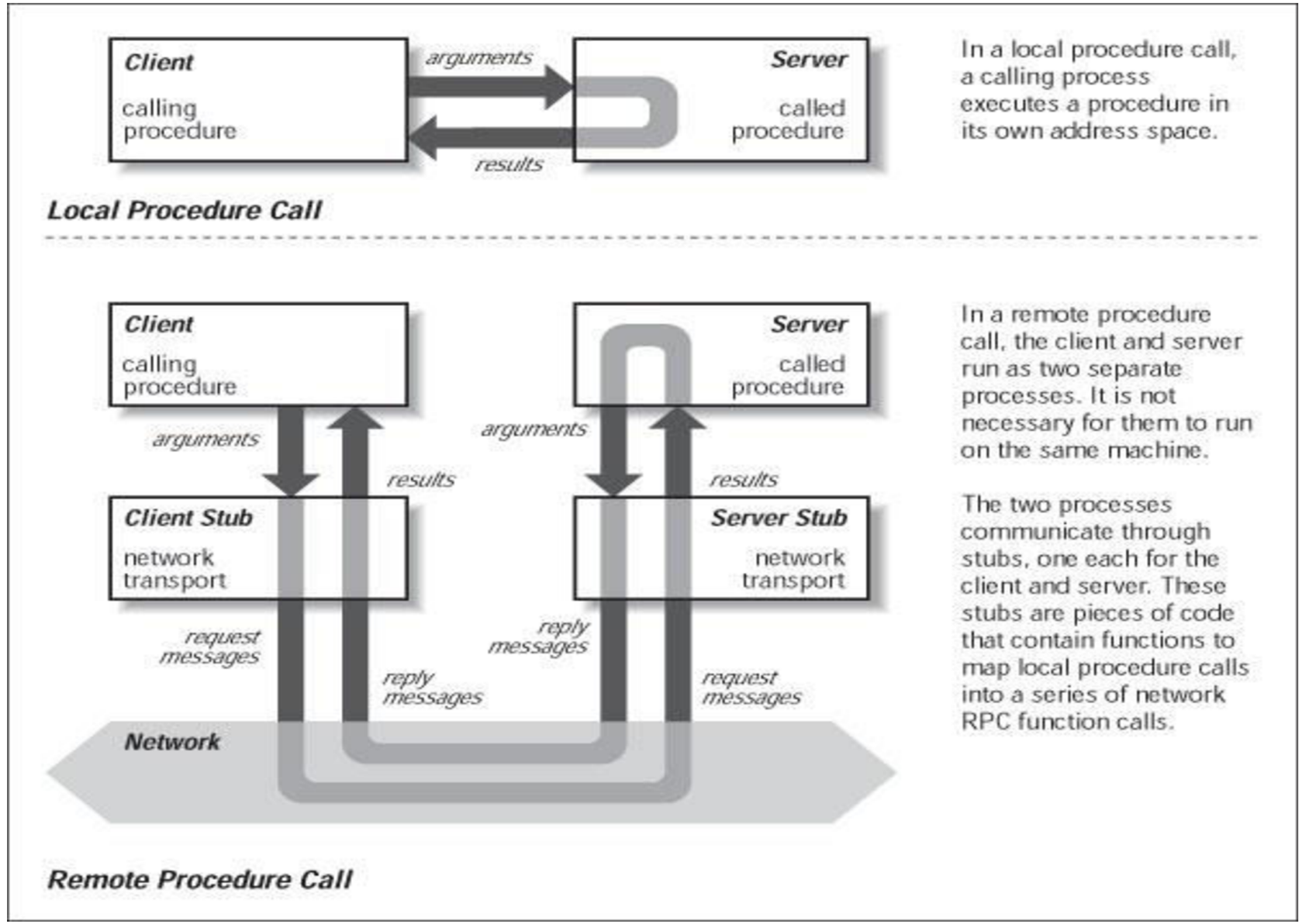
Remote Procedure Call



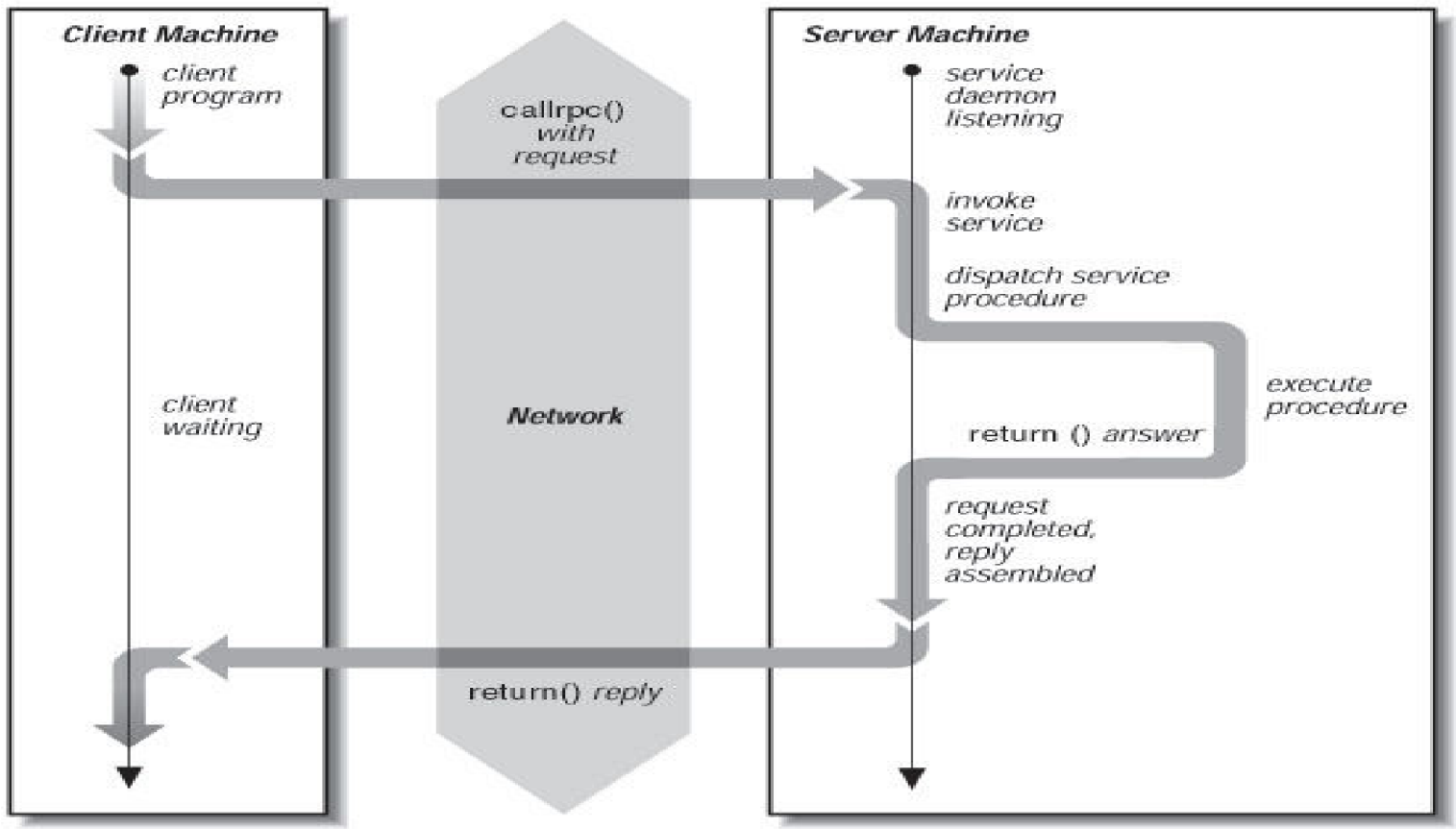
Remote Procedure Call

- An extension of conventional procedure call (used for transfer of control and data within a single process)
- allows a client application to call procedures in a different address space in the same or remote machine
- ideal for the client-server modeled applications
- primary goal is to make distributed programming easy, which is achieved by making the semantics of RPC as close as possible to conventional local procedure call
  - what is the semantics of local procedure call?

Local vs. Remote Procedure Calls



RPC Communication



The client sends out a request over the network. The service daemon is constantly listening for requests. When a request is received, it invokes the service. The appropriate procedure is dispatched. The request is executed and the reply is returned over the network to the client.

The client machine is inactive between the time of the request and when it receives a reply.

The client and server machines may be the same.

RPC System Components

- **Message module**
  - IPC module of Send/Receive/Reply
  - responsible for exchanging messages

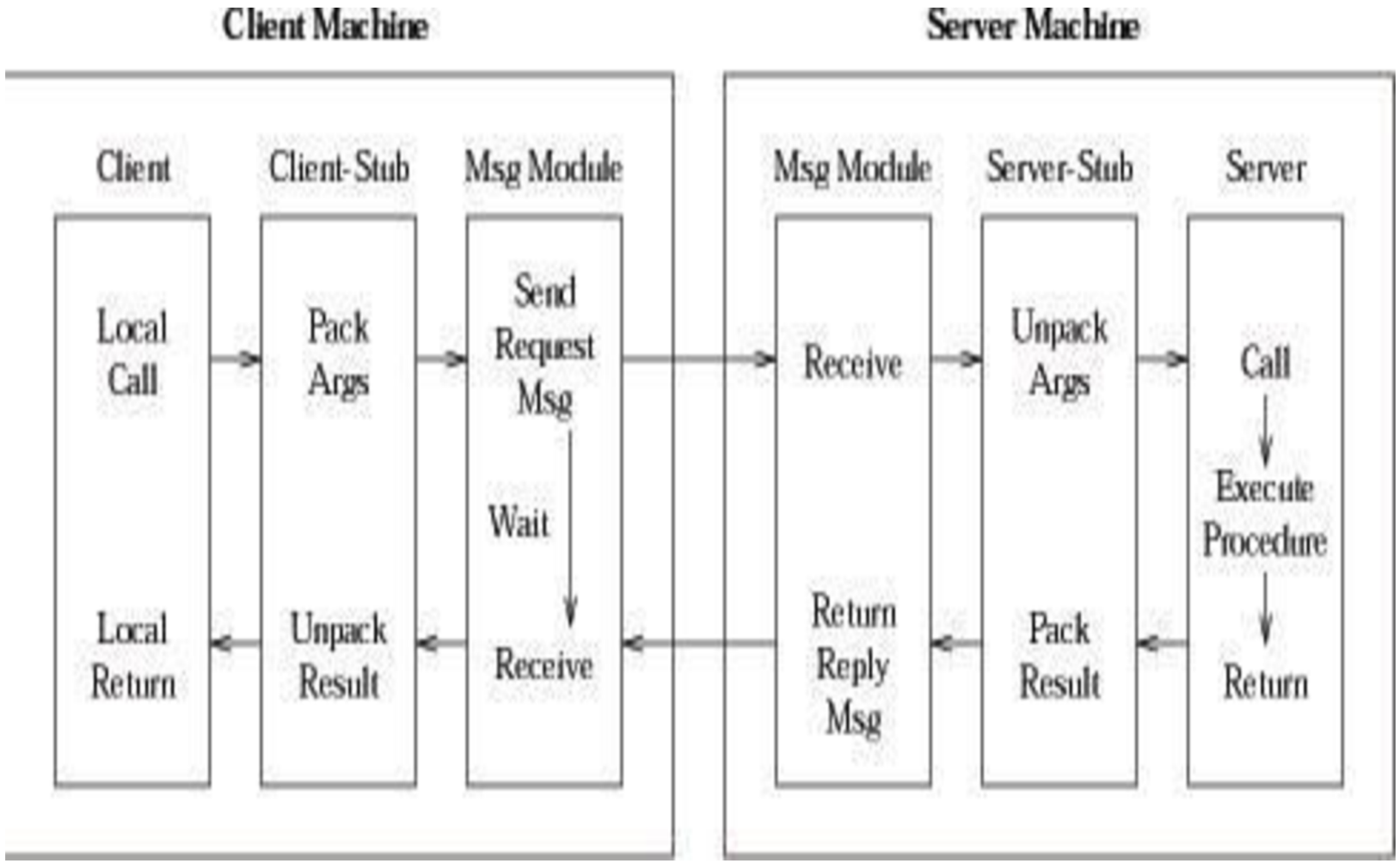
- **Stub procedures** (client and server stubs)
  - a stub is a communications interface that implements the RPC protocol and specifies how messages are constructed and exchanged
  - responsible for packing and unpacking of arguments and results (this is also referred to as “marshaling”)
  - these procedures are automatically generated by “stub generators” or “protocol compilers”

RPC System Components

- **Client stub**
  - packs the arguments with the procedure name or ID into a message
  - sends the msg to the server and then awaits a reply msg
  - unpacks the results and returns them to the client

- **Server stub**
  - receives a request msg
  - unpacks the arguments and calls the appropriate server procedure
  - when it returns, packs the result and sends a reply msg back to the client

RPC System Components and Call Flows



Parameter Passing

- **little endian**: bytes are numbered from right to left

0 3	0 2	0 1	5 0
L 7	L 6	I 5	J 4

- **big endian**: bytes are numbered from left to right

5 0	0 1	0 2	0 3
J 4	I 5	L 6	L 7

How to let two kinds of machines talk to each other?

- a standard should be agreed upon for representing each of the basic data types, given a parameter list (n parameters) and a message.
- devise a network standard or canonical form for integers, characters, Booleans, floating-point numbers, and so on.
- Convert to either little endian/big endian. But inefficient.
- use native format and indicate in the first byte of the message which format this is.

How are pointers passed?

- not to use pointers. Highly undesirable.
- copy the array into the message and send it to the server. When the server finishes, the array can be copied back to the client.
- distinguish input array or output array. If input, no need to be copied back. If output, no need to be sent over to the server.
- still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph.

How can a client locate the server?

- hardwire the server network address into the client.

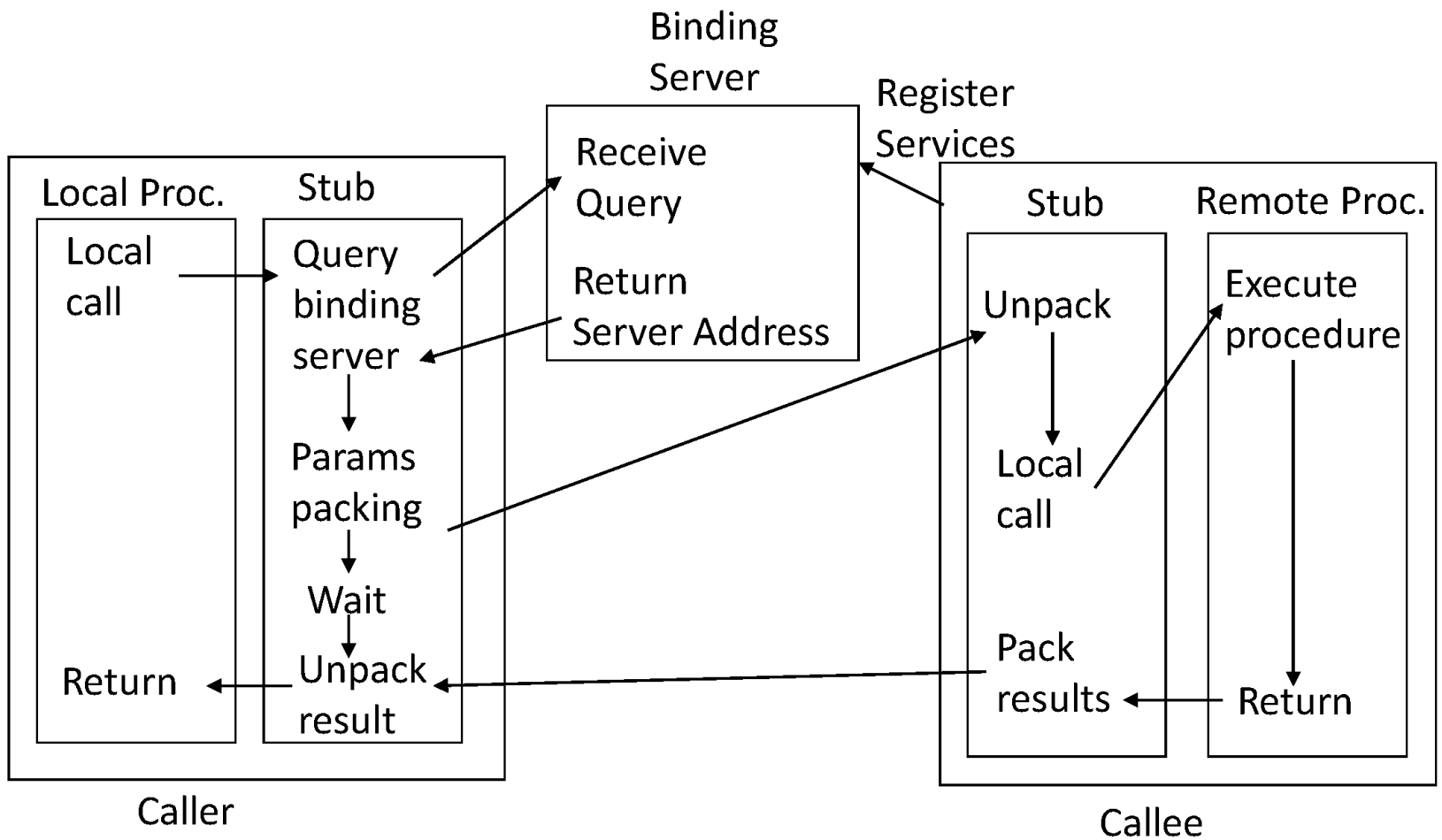
  Disadvantage: inflexible.

- use **dynamic binding** to match up clients and servers.

Dynamic Binding

- Server: **exports** the server interface.
- The server **registers** with a **binder** (a program), that is, give the binder *its name*, *its version number*, *a unique identifier*, and *a handle*.
- The server can also deregister when it is no longer prepared to offer service.

Remote Procedure Call



How the client locates the server?

- When the client calls one of the remote procedure “read” for the first time, the client stub sees that is not yet bound to a server.
- The client stub sends message to the binder asking to **import** version 3.1 of the file-server interface.
- The binder checks to see if one or more servers have already **exported** an interface with this name and version number.
- If no server is willing to support this interface, the “read” call fails; else if a suitable server exists, the binder gives its handle and unique identifier to the client stub.
- The client stub uses the handle as the address to send the request message to.

Advantages

- It can handle multiple servers that support the same interface
- The binder can spread the clients randomly over the servers to even the load
- It can also poll the servers periodically, automatically deregistering any server that fails to respond, to achieve a degree of fault tolerance
- It can also assist in authentication. Because a server could specify it only wished to be used by a specific list of users

Disadvantages

- the extra overhead of exporting and importing interfaces cost time.

Server Crashes

- The server can crash before the execution or after the execution
- The client cannot distinguish these two.
- The client can:
  - Wait until the server reboots and try the operation again (**at least once semantics**).
  - Zero or one execution can take place, if the remote procedure succeeds, exactly one computation has taken place otherwise none (**Exactly once semantics)**
  - Gives up immediately and reports back failure (**at most once semantics**).
  - Guarantee nothing.

Client Crashes

- If a client sends a request to a server and crashes before the server replies, then a computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan**.

Problems with orphans

- They waste CPU cycles
- They can lock files or tie up valuable resources
- If the client reboots and does the RPC again, but the reply from the orphan comes back immediately afterward, confusion can result

What to do with orphans?

- **Extermination:** Before a client stub sends an RPC message, it makes a log entry telling what it is about to do. After a reboot, the log is checked and the orphan is explicitly killed off.
  - Disadvantage: the expense of writing a disk record for every RPC; it may not even work, since orphans themselves may do RPCs, thus creating **grandorphans** or further descendants that are impossible to locate.
- **Reincarnation:**Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message to all machines declaring the start of a new epoch. When such a broadcast comes in, all remote computations are killed.

What to do with orphans?

- **Gentle reincarnation:** when an epoch broadcast comes in, each machine checks to see if it has any remote computations, and if so, tries to locate their owner. Only if the owner cannot be found is the computation killed.
- **Expiration**:Each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum. On the other hand, if after a crash the server waits a time T before rebooting, all orphans are sure to be gone.
- None of the above methods are desirable.

Implementation Issues

- the choice of the RPC protocol: connection-oriented or connectionless protocol?
- general-purpose protocol or specifically designed protocol for RPC?
- packet and message length
- Acknowledgements
- Flow control

overrun error: with some designs, a chip cannot accept two back-to-back packets because after receiving the first one, the chip is temporarily disabled during the packet-arrived interrupt, so it misses the start of the second one.

How to deal with overrun error?

- If the problem is caused by the chip being disabled temporarily while it is processing an interrupt, a smart sender can insert a delay between packets to give the receiver just enough time.
- If the problem is caused by the finite buffer capacity of the network chip, say n packets, the sender can send n packets, followed by a substantial gap.

Chapter 2: A Model of Distributed Computations

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

A Distributed Program

- A distributed program is composed of a set of n asynchronous processes, p_1, p_2, ..., p_i, ..., p_n.
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let C_{ij} denote the channel from process p_i to process p_j and let m_{ij} denote a message sent by p_i to p_j.
- The message transmission delay is finite and unpredictable.

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, namely, internal events, message send events, and message receive events.
- Let e_i^x denote the xth event at process p_i.
- For a message m, let $send(m)$ and $rec(m)$ denote its send and receive events, respectively.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

A Model of Distributed Executions

- 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, ..., e_i^x, e_i^{x+1}, ... and is denoted by $\mathcal{H}_i$ where

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

  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.

A Model of Distributed Executions

- 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

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

- Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events.

A Model of Distributed Executions

- The evolution of a distributed execution is depicted by a space-time diagram.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.
- Since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to denote it as a dot on a process line.
- In the Figure 2.1, for process p_1, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

A Model of Distributed Executions

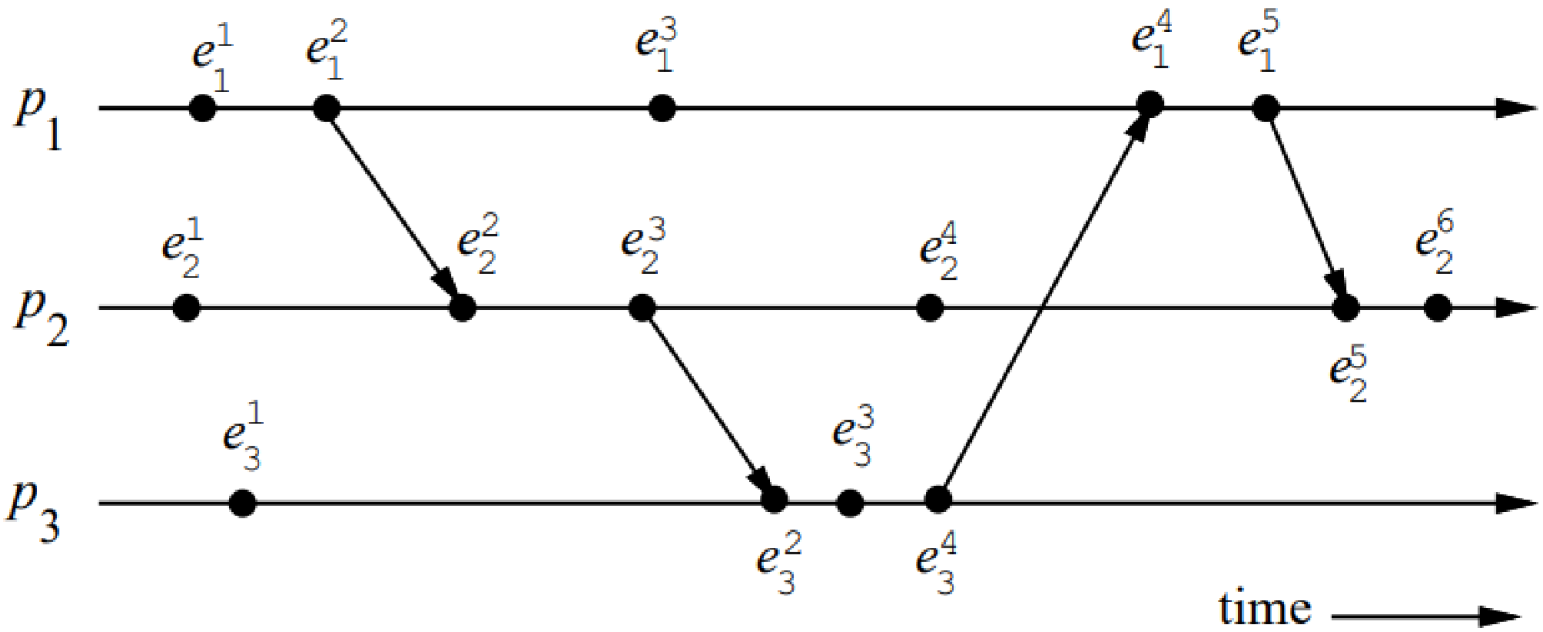


Figure 2.1: The space-time diagram of a distributed execution.

A Model of Distributed Executions

Causal Precedence Relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let $H = \cup_i h_i$ denote the set of events executed in a distributed computation.
- Define a binary relation $\rightarrow$ on the set H as follows that expresses causal dependencies between events in the distributed execution.

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

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

A Model of Distributed Executions

. . . Causal Precedence Relation

- Note that the relation $\rightarrow$ is nothing but Lamport's "happens before" relation.
- For any two events 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 (along increasing time) in the space-time diagram that starts at e_i and ends at e_j.)
- For example, in Figure 2.1, $e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$.
- The 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.
- For example, in Figure 2.1, event e_2^6 has the knowledge of all other events shown in the figure.

A Model of Distributed Executions

. . . Causal Precedence Relation

- For any two events e_i and e_j, $e_i \not\rightarrow e_j$ denotes the fact that event e_j does not directly or transitively dependent on event e_i. That is, event e_i does not causally affect event e_j.
- In this case, 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, in Figure 2.1, $e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$.

Note the following 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, $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.

A Model of Distributed Executions

Concurrent events

- 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 (denoted as $e_i \parallel e_j$).
- In the execution of Figure 2.1, $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$.
- The relation $\parallel$ is not transitive; that is, $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$.
- For example, in Figure 2.1, $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$, however, $e_3^3 \not\parallel e_1^5$.
- 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 $e_i \parallel e_j$.

A Model of Distributed Executions

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 connotation 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 occured at the same instant in physical time.

Models of Communication Networks

- There are several models of the service provided by communication networks, namely, FIFO, Non-FIFO, and causal ordering.
- In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the 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.

Models of Communication Networks

- The "causal ordering" model is 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.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that CO $\subset$ FIFO $\subset$ Non-FIFO.)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

Global State of a Distributed System

"A collection of the local states of its components, namely, the processes and the communication channels."

- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that or receives the message and the state of the channel on which the message is received.

. . . Global State of a Distributed System

Notations

- LS_i^x denotes 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.
- Let $send(m) \leq LS_i^x$ denote the fact that $\exists y{:}1 \leq y \leq x :: e_i^y = send(m)$.
- Let $rec(m) \not\leq LS_i^x$ denote the fact that $\forall y{:}1 \leq y \leq x :: e_i^y \neq rec(m)$.

. . . Global State of a Distributed System

A Channel State

- The state of a channel depends upon the states of the processes it connects.
- Let $SC_{ij}^{x,y}$ denote the state of a channel C_{ij}.

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} | \ send(m_{ij}) \leq e_i^x \bigwedge rec(m_{ij}) \not\leq e_j^y\}$$

Thus, channel state $SC_{ij}^{x,y}$ denotes all messages that p_i sent upto event e_i^x and which process p_j had not received until event e_j^y.

. . . Global State of a Distributed System

Global State

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

$$GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$$

- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)

. . . Global State of a Distributed System

A Consistent Global State

- Even if the state of all the components is not 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 a state should not violate causality – an effect should not be present without its cause. A message cannot be received if it was not sent.
- 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.
- 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

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

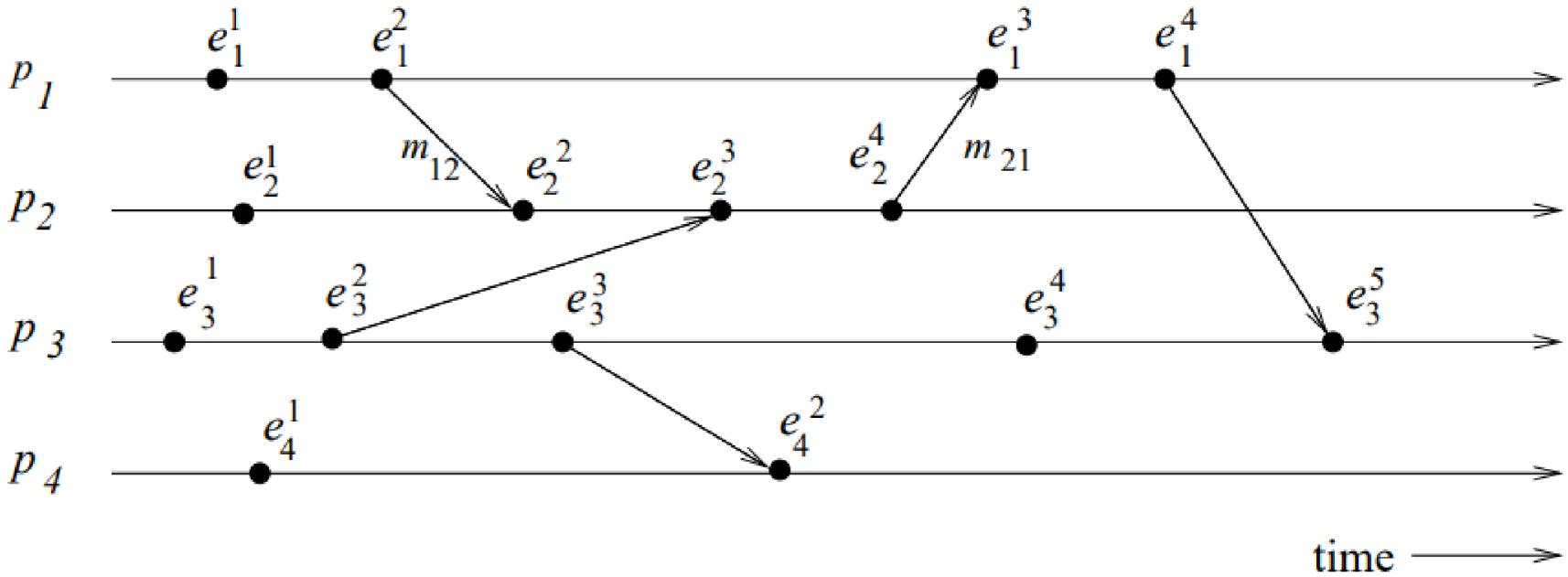
- That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.

. . . Global State of a Distributed System

An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



. . . Global State of a Distributed System

In Figure 2.2:

- A global state $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ 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.
- A global state GS_2 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}.

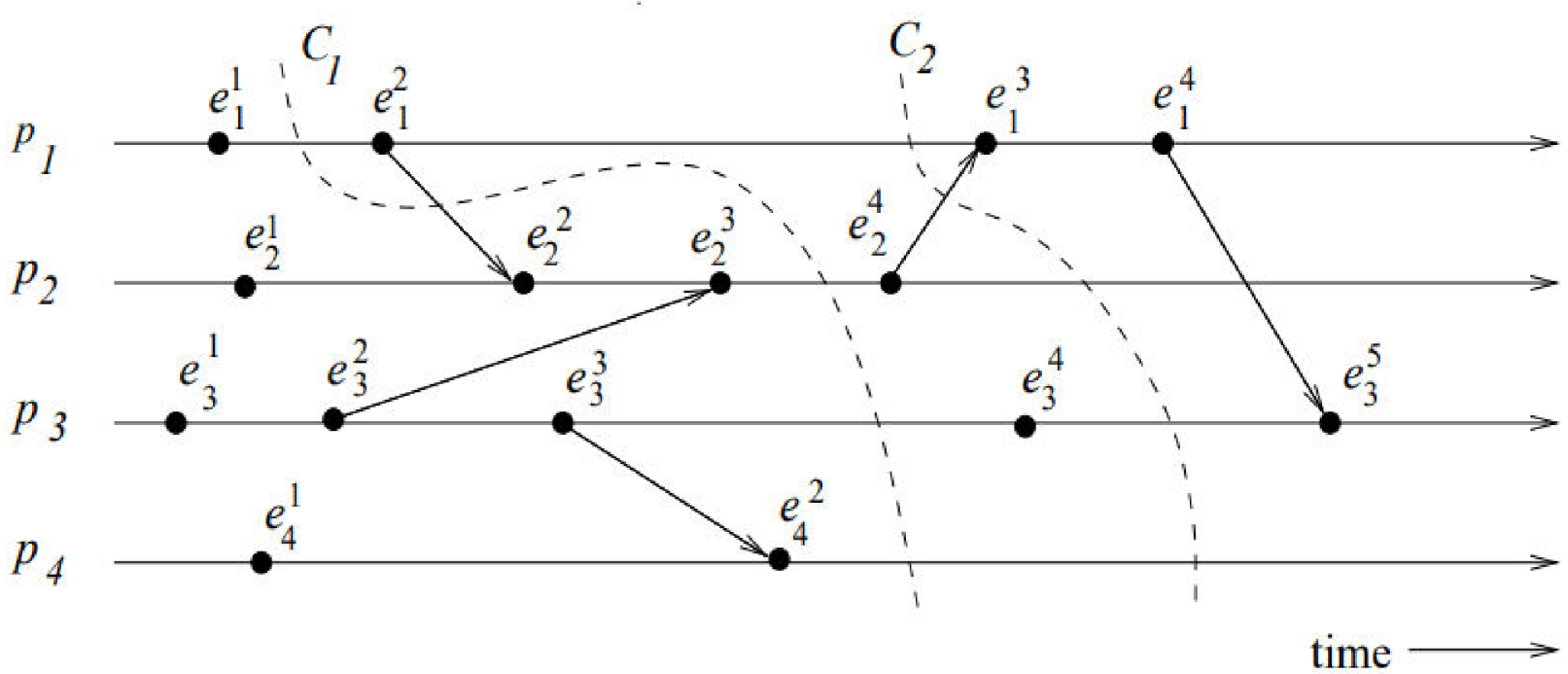
Cuts of a Distributed Computation

"In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line."

- A cut slices the space-time diagram, and thus the set of 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, let PAST(C) and FUTURE(C) denote the set of events in the PAST and 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 in a space-time diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

. . . Cuts of a Distributed Computation

Figure 2.3: Illustration of cuts in a distributed execution.



. . . Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure 2.3, cut C_2 is 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. (In Figure 2.3, cut C_1 is an inconsistent cut.)