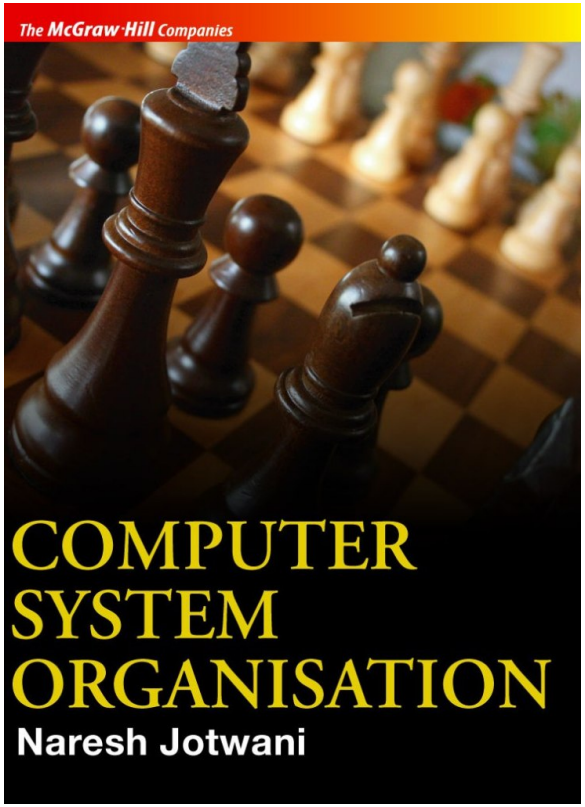
Mc Graw Hill Education

COMPUTER SYSTEM ORGANISATION

Naresh Jotwani

PowerPoint Slides



CHAPTER 14

MULTIPROCESSOR SYSTEMS

Introduction

- A computer system can be provided with multiple processors so that it can process data at a higher effective rate.
- On such a multiprocessor system, each processor can independently execute the instructions of a program, in parallel with other processors.
- On such a computer system, each processor in has its own instruction stream and data stream.
  - Therefore such systems can be classified as *multiple instruction stream, multiple data stream* (MIMD) systems.

- Depending on whether main memory is shared, multiprocessor systems may be classified as:

- (i) *Shared memory multiprocessors*:
  - Processors and main memory communicate over the common processor-memory bus.
  - Communication between running programs can take place through the use of shared memory locations.

- (ii) *Cluster computing and distributed computing*:
  - Each processor has its own main memory, making up a computing element.
  - Processor-memory bus does not extend from one computing element – or 'node' – to another.
  - Communication between computing nodes takes place using communication capability provided amongst the nodes.

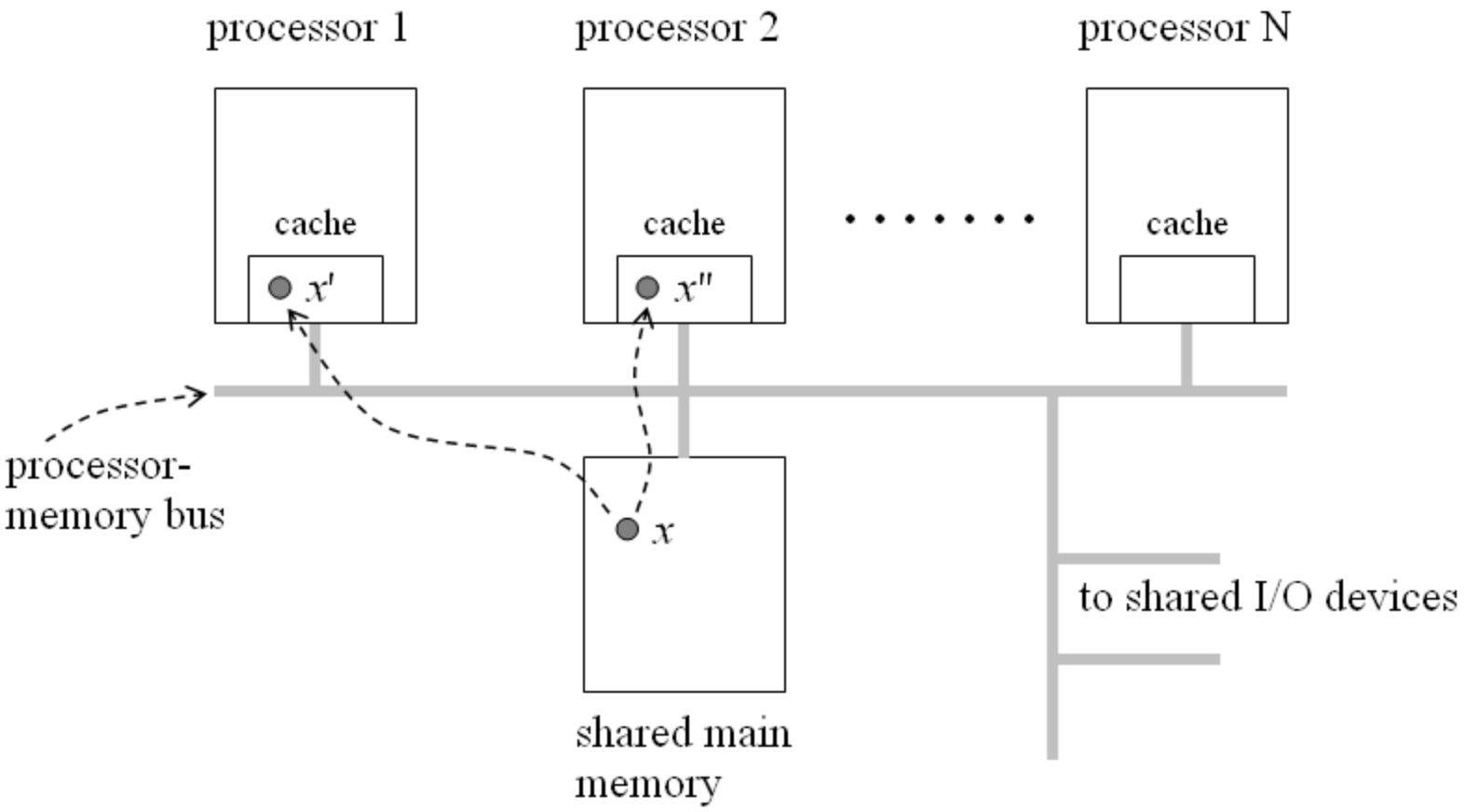
Shared memory multiprocessors

- Typical organization of a *shared memory multiprocessor* system is shown in the next figure.

- The *N* processors shown are assumed to be identical, in terms of their instruction set, computing power, external bus interface, and so on.

- Processors access common shared main memory for instructions and data.

- *Processor-memory bus* is a crucial system component, since all *read* and *write* operations are performed between processors and main memory over this bus.



- For each *read* and *write* operation, if the *N* processors access main memory over the common bus, then *contention* between processors for access to the bus will become a severe performance bottleneck.

- In a single clock cycle of the bus, only one processor can use the bus for a *read* or *write* access to main memory.

- If two or more processors need to use the common bus at the same time, a *bus arbitration* mechanism ensures that one processor uses the bus, while the others wait for one or more bus cycles

- If a processor is waiting to use the bus, its processor cycles are wasted for the duration of the wait.

- If memory requests of all *N* processors require use of the bus, then such wasted processor cycles are inevitable.

Example

- For every instruction executed by a processor, one memory access is required in instruction fetch phase. Assume that, on average, 40% of the instructions require a second memory access in execute phase.
  - Recall that the second memory access is required for instructions such as load, store, function call, and return.

- Thus the average number of memory accesses required per instruction is 1.4, for each of *N* processors.
  - If each processor executes one instruction per clock cycle, the average number of memory references per clock cycle is 1.4x*N*.
  - For $N = 4$, this is 5.6 memory references per clock cycle, which is not sustainable over the bus or in main memory.

- Thus *contention* for shared bus and main memory becomes significant with even a small number of processors.

- Solution: If separate cache memory is provided on each processor, then main memory access by a processor is required only on a cache miss.
  - Example: If we assume the cache miss rate to be 5%, then the number of memory accesses required is reduced by a factor of 20 − and the shared memory multiprocessor architecture with a common bus becomes viable.
  - Therefore each processor in a shared memory multi-processor system must have its own cache.
- Of course each processor should also be built to exploit instruction level parallelism (ILP) and thread level parallelism (TLP), as seen in the previous chapter.

- In such a shared memory multiprocessor system, the common processor-memory bus plays the central role in achieving effective utilization of all system resources.

- Another important issue on such systems is that of *load balancing*.
  - i.e. as far as possible, dividing the total processing workload equally across the *N* processors.
  - The workload consists of both operating system and application programs.
  - The potential benefits of having multiple processors will not be realized if, for example, one processor becomes over loaded while another one spends its time in ‘idle’ mode.

- *Process migration* -- If processor A is over-loaded and processor B is under-loaded, can we *migrate* one or more processes from A to B?

- To achieve process migration, we need to
  - (i) identify the process to be migrated, and
  - (ii) transfer the full context of that process – i.e. registers, memory map, and so on – from processor A to B.

- *Multi-core processors* (two or more processors on a single VLSI chip) have recently been introduced.
  - Each processor has its own L1 cache,
  - System configuration is a shared-memory multiprocessor.

- Multi-core processors are developed to squeeze more processing power out of available VLSI technology.

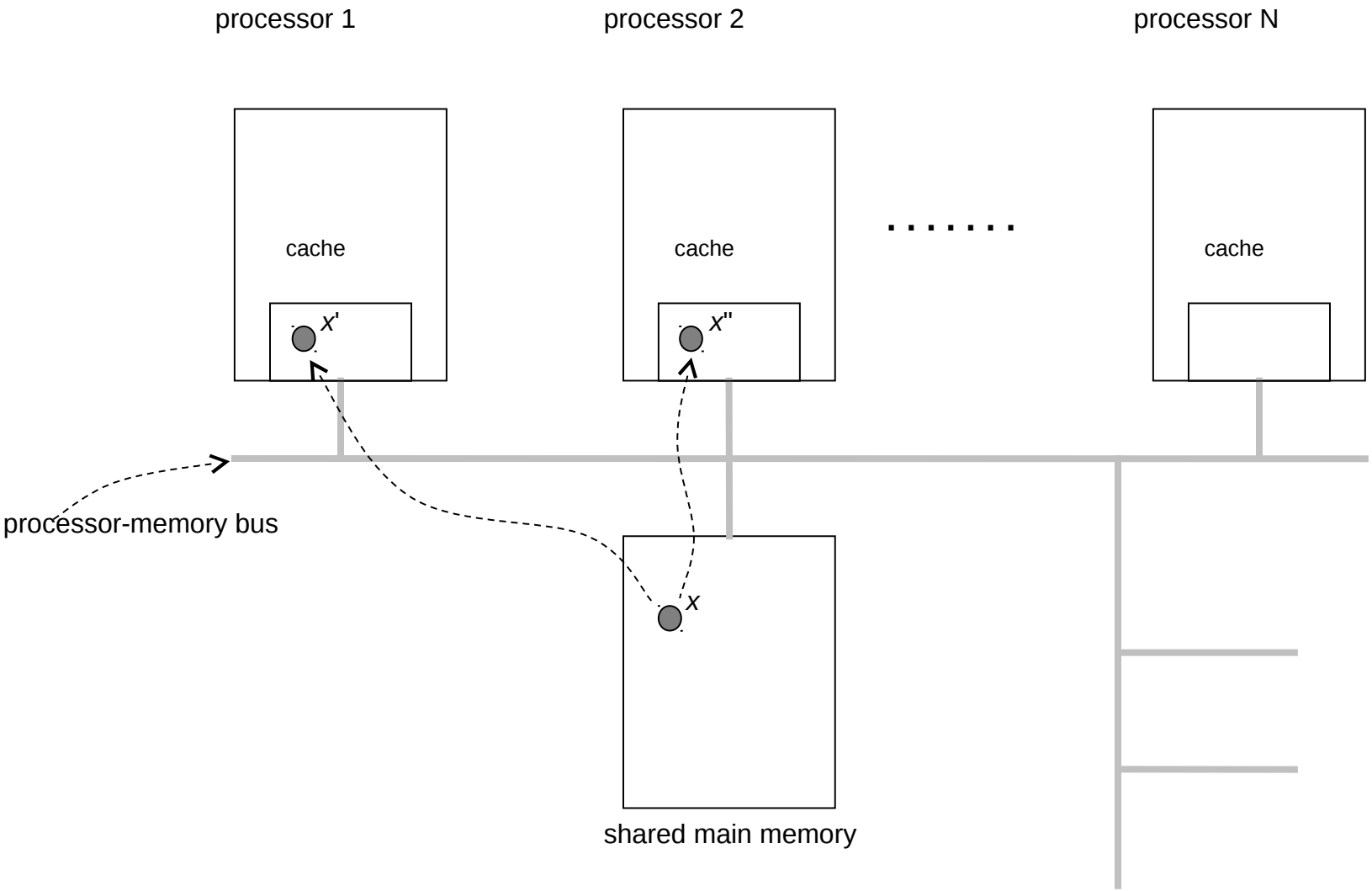
Cache Coherence

- When each processor has its own cache memory, a difficulty is created when programs running on two or more processors share items of data in main memory.

- See the next figure. Data item *x* is shared by the two programs running on processors 1 and 2, respectively.

- If both these programs make use of *x* over a period of time, a copy of *x* will remain in the cache memory of each of the processors. In the figure, these two copies of *x* have been labeled as *x*' and *x*'' respectively.

- Suppose the program on processor 1 modifies the value of this shared variable. The change is made in its cache memory copy of the variable, labeled *x*'.


processor 1
processor 2
processor N
cache
cache
cache
x'
x''
x
processor-memory bus
shared main memory

- Cache memory mechanism causes the main memory copy *x* of the variable to be modified accordingly, either immediately or when *x*' is removed from cache.

- The program on processor 2 is also sharing this variable.
  - So, after *x*' is modified, we must reflect this change in copy *x*" of the variable as well, otherwise the program on processor 2 will see the old value of the variable.

- Ideally, the three copies *x*, *x*', and *x*" of the variable should reflect the same value at all times.

- This means that whenever any processor changes the value of the shared variable in its copy, the change should be reflected in all copies.

- Therefore *cache coherence* must be maintained in any shared memory multiprocessor system. One way:

- Cache coherence is lost only when a processor modifies the value of a shared variable in its cache – e.g. when processor 1 modifies *x*'.
  - When this happens, all other copies of *x*, such as *x*" in processor 2, are said to become *invalidated*.
  - When *x*" is thus invalidated, and processor 2 tries to read its value, the cache memory sub-system should return a *cache miss*, since the value of *x"* is invalid.

- How does a cache sub-system know that a such a value has become invalidated?
  - Cache memory sub-system on each processor *snoops* on the common processor-memory bus – i.e. monitors the bus – for any memory *write* operations.

- When a *write* operation is detected on the bus, the memory address on the bus is matched with the cache content using single-cycle associative search.
  - When the cache detects that a copy of data shared by it is being modified (i.e. written into main memory) by another processor, it *invalidates* its own copy.
- The next reference to this variable generates a cache miss, and forcing access from main memory.
  - We have assumed that the cache memory is working in *write-through* mode, i.e. each memory write operation results in the main memory copy of the data being modified immediately.
- This mechanism is known as *snooping cache*.

- A possible improvement is this:
  - When a cache memory detects on the bus a memory write operation on a shared data item, it picks up the value from the bus and updates its own copy.

  Thus, its copy of data item is updated rather than invalidated.

- One other approach makes use of a *directory* of the aggregate cache content of all the *N* processors.
  - The directory is built using fast associative memory, maintains the status of all the cache content, and updates it on all *writes* to main memory.
  - Thus, instead of each cache memory snooping on the bus, the directory provides a centralized mechanism for maintaining cache coherence.

Synchronization and Arbitration

- A multiprocessor system, with explicit parallelism between processors and processes, must be designed to support inter-process synchronization.
- A fundamentally different kind of processor instruction is provided to serve as the 'building block' of synchronization.
- This instruction – *test and set lock* (TSL) – carries out a *read* and a *write* operation on main memory as one *indivisible* combination of the two operations.

- An interrupt signal arriving during the execution of TSL is only acted upon after TSL completes its execution.

- Let LOCK be a memory location shared by two or more processes. Consider the following instruction:

```
TSL  LOCK, R0
```

- The value in memory location LOCK is read into R0, and the value ‘1’ is written into the same memory location.

- After TSL, the program tests the value in the register. Value ‘0’ indicates that the shared data item or resource is *unlocked*, i.e. the program can proceed to perform the required operations on it.

- Independent of the value read into R0, TSL sets the value of memory variable LOCK to ‘1’. Any other process subsequently checking its value will see ‘1’ – i.e. it will find the shared data item or resource *locked*.

- The process which locks the data or resource unlocks it by setting the memory variable LOCK to ‘0’.

- The initial value of LOCK is set to ‘0’. At most one process at a time ‘holds the lock’ for access to the shared data item or resource.

- In its *execute phase*, TSL requires two bus cycles over the processor-memory bus – one for memory *read* and one for memory *write*.

- For TSL to be indivisible, the processor-memory bus should not be used by another processor while one processor is executing TSL. The bus should be locked during the *read* and *write* memory cycles of TSL.

- A closely related issue is that of *bus arbitration*. Suppose two or more of the *N* processors need to make use of the bus in the same clock cycle.

- The address and data lines of the bus are designed to be used by only one processor in a given clock cycle.

- Therefore only one processor must be granted use of the bus, and the other processor or processors must wait.

- The bus arbitration mechanism works in hardware. If multiple processors make *bus request* at the same time, only one of them receives the *bus grant* signal.

- Processor cycles are wasted when a processor has to wait for use of the bus – thus there is a negative impact of bus contention on system performance.

- When multiple processors contend for the bus, it may not matter which of them uses the bus first − as long as only one processor at a time does so.

- Since bus contention has a negative impact on system performance, it must be reduced by providing each processor with its own cache memory.

- Bus arbitration mechanism resolves the inevitable contention for the bus which remains even after each processor is provided with its own cache memory.

Message-passing model

- If processors in the system do not share common main memory, each processor has its own local memory for instructions & data, and I/O capability to send and receive *messages* to/from other processors.
- A processor with its local memory and I/O capability is then referred to as a *processing node*, or simply *node*.
- To identify communicating nodes in such a system, each node is provided with a unique *node ID*.
- An application running on such a system is designed as a set of processes running on the *N* processing nodes, exchanging *messages* amongst each other as required by the application.

- Such systems offer the important advantage of *scalability* – i.e. this model of multiprocessing is effective even for larger values of *N*.

- Sending and receiving messages are *higher level* operations – in the sense that many machine instructions must be executed to accomplish a *send* or a *receive*.

- Typically, *send* and *receive* are functions which must be called by a program as needed.

- These functions are made available to the programmer as part of the operating system. Arguments to these functions include *message address and length*, *receiving node ID*, and so on.

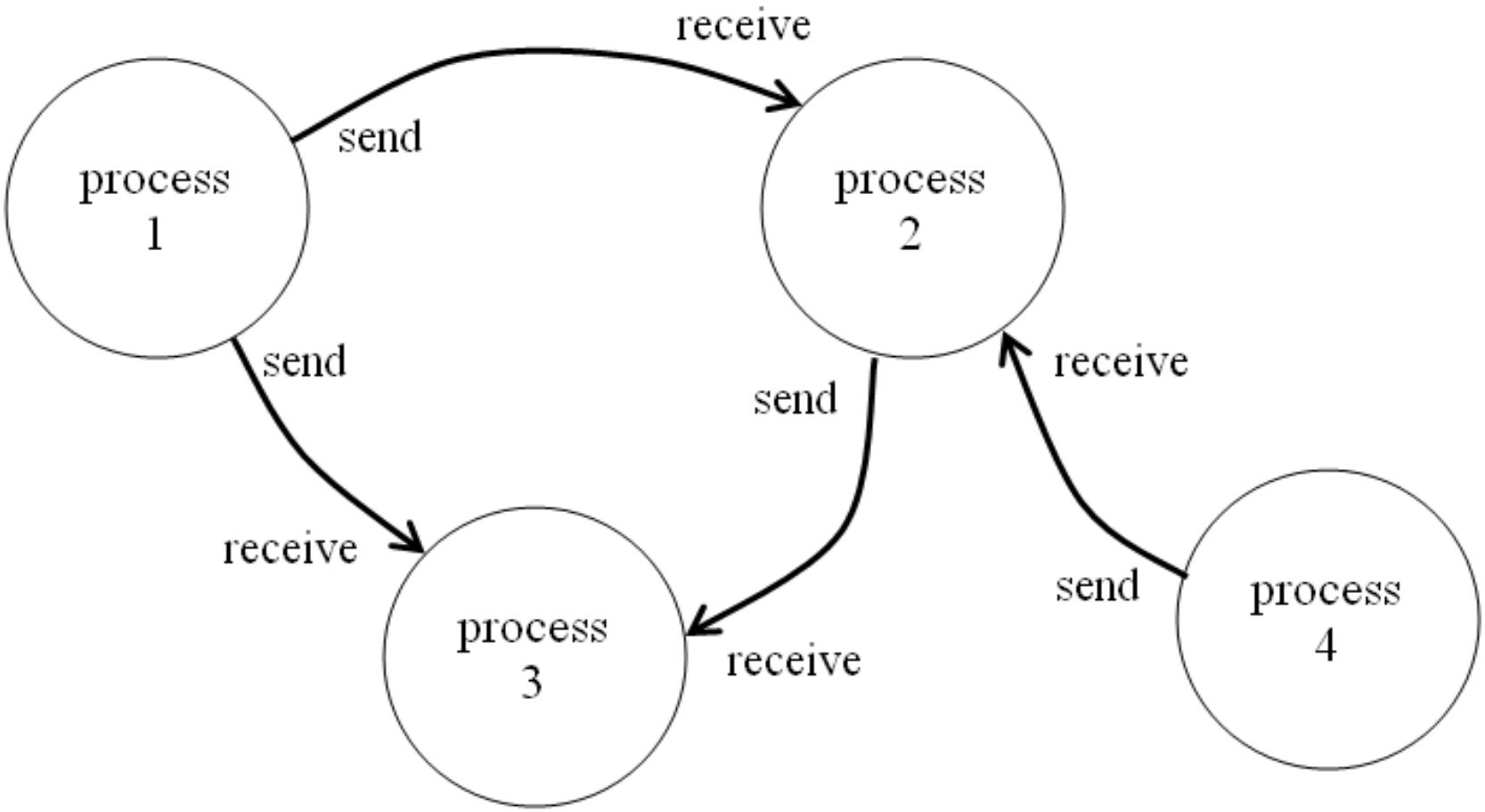
- Since *send* and *receive* are functions rather than machine instructions, on such a system we do not say that a processor executes a *send* or a *receive* operation.

- A processor executes a machine instruction, but a *process* invokes and executes functions such as *send* and *receive*.

- In other words, the entities performing *send* and *receive* are not processors but *processes*.

- We can thus visualize our multiprocessing system as a *set of processes* which send and receive messages amongst one another.

- Example of such processing is shown in the next figure, with a set of four processes.


receive
send
process
1
process
2
send
send
receive
receive
process
3
receive
send
process
4

- In moving our perspective from *processor* to *process*, we perform an act of *abstraction*.
  - We now speak of functions such as *send* and *receive*, rather than machine instructions such as LOAD and STORE.
  - And we stop thinking in terms of physical attributes of processors such as address size, word size, or registers.
- Of course we assume that functions *send* and *receive* can be programmed on every processor in the system.

  Beyond that, we need not even assume that the processors at various nodes in the system are identical.
- Abstraction also means that we think about *processes* in terms of the functions they perform, rather than machine instructions they execute.

- As long as processes *send* and *receive* messages correctly, the system also performs its function correctly.

- Application is made up of the functions performed by its constituent processes; these functions include *send* and *receive* operations between processes. This is the *message-passing model* of parallel processing.

- The degree of true parallelism in the system is N, the number of processing nodes. The number of *processes*, say M, need not be equal to N.

- We can say that $1 \leq N \leq M$, since the system needs at least one processing node, but not more than M. If two or more processes are running on the same node, then standard multiprogramming accommodates them.

- The overall *throughput* of such a system depends on the number and processing power of the nodes being put to work, but the *correctness of results* does not.

- The type of communication provided between nodes may be different.
  - Two nodes in the same room may communicate over a very fast link; those separated by a few hundred kilometers may communicate over the Internet.
  - As long as *send* and *receive* functions work correctly between processes, correctness of the system is assured.

- Thus message-passing provides us with a very general, robust, and consistent model of parallel processing.

- The model is so general that, depending on specific system features, two broadly different types of systems can be defined using this model:

  - (i) If the *N* processing nodes in the system are of identical type, and are interconnected locally over high-speed communication links, the resulting system is known as a *computing cluster.* And this model of computing is known as *cluster computing*.

  - (ii) If the processing nodes are not necessarily of one type, and are interconnected using network links of variable speeds over larger distances, the model of computing is known as *distributed computing*.

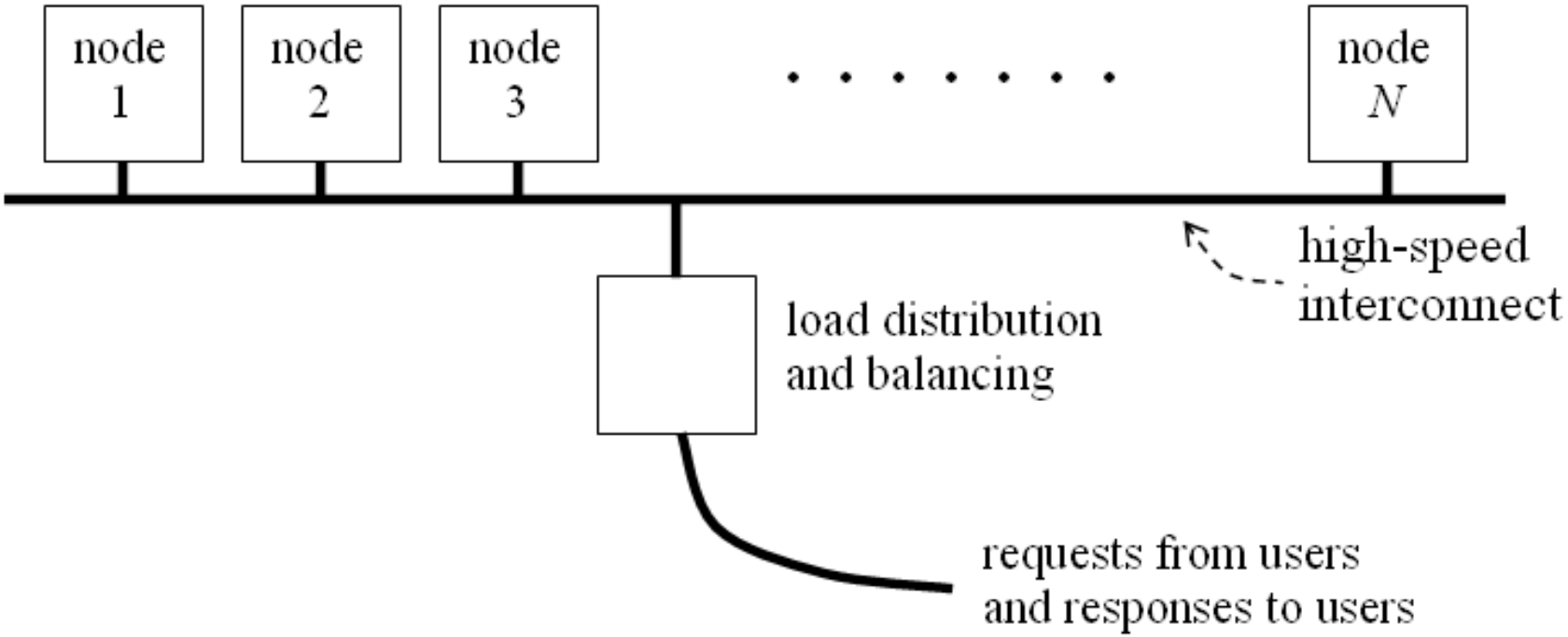
- Both *cluster computing* and *distributed computing* are variants of MIMD multiprocessing, without common shared memory between nodes.

- But what happens on such a system when $N = 1$, i.e. there is only one processing node?
  - We then have M processes running in pseudo-parallelism on the single node. But, even on such a uni-processor system, the M processes can use message-passing for synchronization.

- Even on a shared memory system with $N > 1$, processes can use message-passing for synchronization.
  - In such cases, synchronization based on lock variables in main memory tends to be more efficient than message-passing.

Cluster Computing

- *N* identical processing nodes connected over a high-speed network, which usually spans a limited distance.
- Each node consists of components – i.e. processor, memory, network adaptor – which are essentially the same as those used in desktop computers or servers.
- Keyboard and display are required only on nodes through which a user interacts with the system.
- The next figure illustrates schematically this version of multiprocessing architecture.



- This architecture has the advantage of being highly *scalable* – i.e. it can be built with just a few processing nodes in a small room, or with many hundreds of processing nodes for large applications.

- In the ranking of the most powerful computers around the world, today cluster computing systems make up the single largest fraction.

- In fact today such systems have completely replaced the specialized and expensive supercomputers which were in use until a couple of decades ago.

- In a system of the type shown, the *N* processors may execute the same application program – but each processor operates on a different 'slice' of the aggregate data being processed by the system.

- Message-passing provides the required synchronization between the processes which make up the application.

- Typically, a separate processing node can provide the coordination needed – e.g. handling system functions such as user requests and responses, load distribution, load balancing, and so on.

- With message-passing architecture, the interconnected processors are relieved from the restrictions imposed by a rigid communication structure.

- Therefore it becomes easier to map the topology of any computational problem – i.e. any user application – to the computing cluster.

- This makes the system much more *versatile* in handling different types of applications, and explains the success of this model of parallel processing in recent years.

- Apart from many scientific applications of such systems, popular web-based applications require 'back-end' servers with huge processing capacity and secondary storage, to handle thousands of requests per second.

- Such servers are built using a number of processors in a computing cluster. Processing load consists of the large number of user requests which must be serviced with acceptable response time. These requests are processed in parallel across the processors.

- Thus *scalability* and *versatility* are major advantages of this model of multiprocessing.

- In addition, this model offers two other significant advantages which explain its huge success – *low cost per processing node* and *fault-tolerance*.

Interconnection Structures

- In a computing cluster, if all nodes are connected using a single communication link, at a given time a single node can send a message to one or more recipient nodes.
- A message sent on the link is seen by all the nodes, but only the intended recipients pick up the message. Such a link is known as a *broadcast link*.
- Over such a *broadcast link*, at a given time, any processing node *i* can send data to any other node *j* (or even a group of nodes) – regardless of the physical location of the nodes on the network.
- If node *i* is sending data to node *j*, then at the same time another node *m* cannot send data to node *n* – since all the nodes share the common communication link.

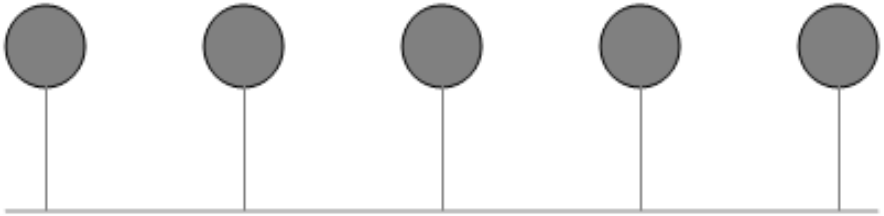
- This restriction does not cause a performance limitation if the link data transfer capacity is sufficiently high.
- A broadcast interconnect does not place any restriction on the pattern of communication between nodes.
- The resulting flexibility in the system is a great advantage – i.e. different applications can make use of the broadcast link in different ways.
- Many computing clusters in use today make use of a broadcast interconnect amongst processing nodes.
- However, alternatives have also been sought to the use of a single, common broadcast link amongst the *N* processing nodes of the computing cluster.

- An alternative to a broadcast link is a *point to point link* − a link with exactly two nodes connected to its two ends.

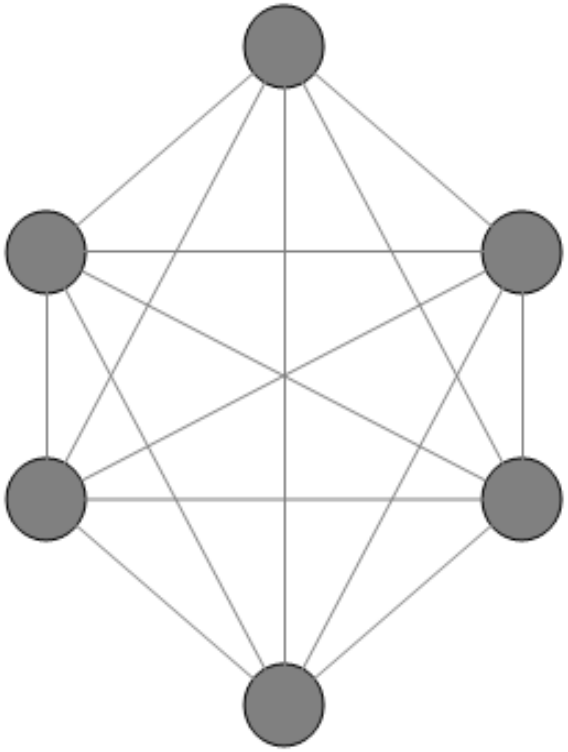
- A message sent along such a link by the node at one end, say node A, is only received by the node at the other end, say node B.
  - We also assume that each such point to point link is *full duplex*, i.e. it can simultaneously carry a message from node A to node B, and from node B to node A.

- With *N* processing nodes in a computing cluster, and with the use of point to point links between pairs of nodes, several types of interconnection schemes can be devised.

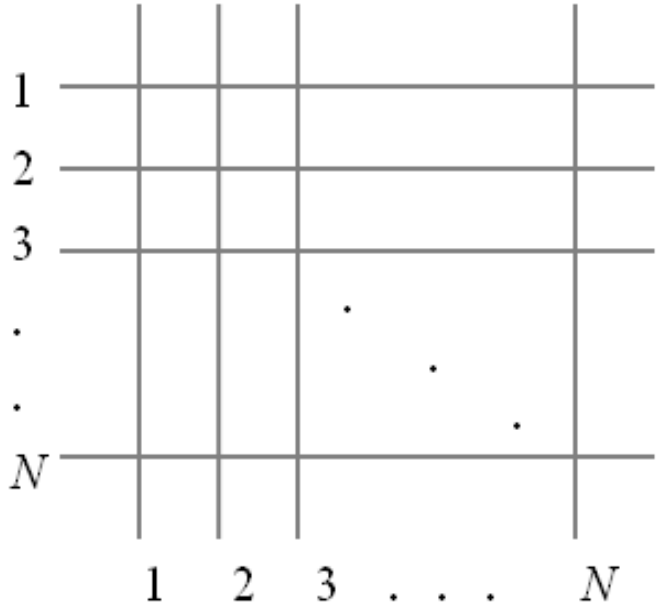
- The next figure shows some possible types of interconnection. Part (a) of the figure shows a broadcast interconnection, of the type discussed above.



(a) a broadcast medium



(b) complete graph with $N = 6$

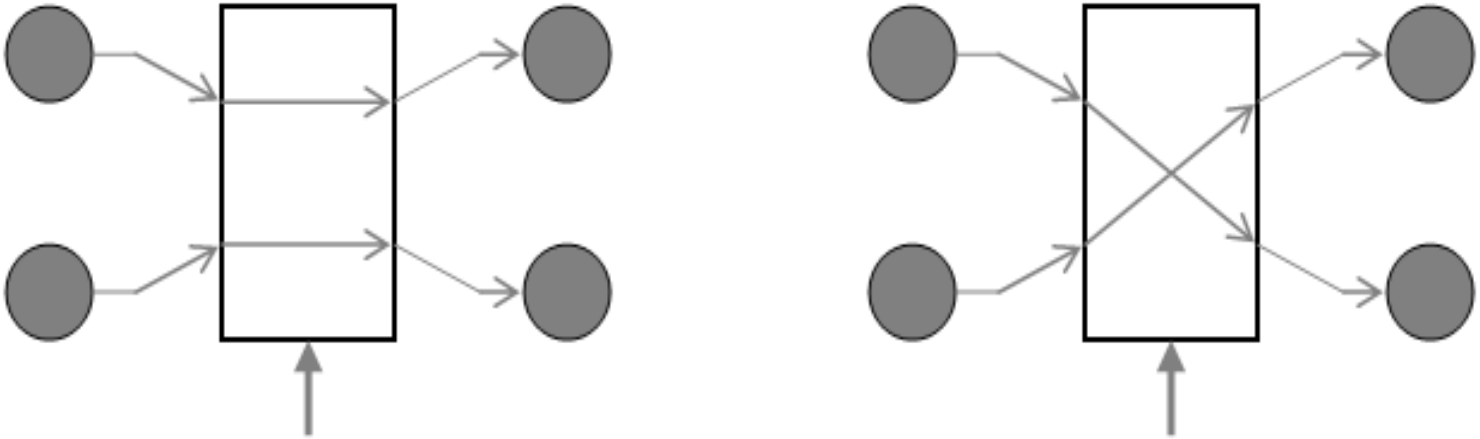


(c) Crossbar switch

- In part (b) of the figure, we see a *fully connected cluster* of N = 6 nodes, in which there is a point to point link between every pair of nodes.
  - It is a simple exercise to verify that such a cluster contains a total of Nx(N-1)/2 links.
  - At one time, any node can send or receive a message to/from any other node, since there is no question of any contention amongst nodes for access to a link.
  - But such a configuration is impractical because it is not *scalable* since, for large N, the number of links required grows as N^2.

- In part (c) of the figure we see a *cross-bar switch* to which the *N* nodes are connected.
  - Note that the same *N* nodes occur along the rows of the switch (shown at the left), and along the columns (shown at the bottom).
  - At any one time, any node *i* can be connected to any node *j*. But, at that time no other node can be connected to either *i* or *j*.
  - Thus, at one time, *N*/2 connections are possible, between distinct pairs of nodes.
- The pattern of interconnections amongst nodes can be switched electronically as needed.

- Figure below shows a 2x2 *switch*, i.e. a switch with two input links and two output links.
  - At one time, each input link can be connected to exactly one output link. The two possible interconnections between inputs and outputs are shown in the figure.
  - The arrow shown at the bottom of the rectangle is the control signal which determines the interconnection.
- Multiple such switches can be used for connecting *N* processing nodes.



- In a computing cluster, the processing nodes do not share common memory. Communication between nodes takes place in the form of network I/O.

- A high speed *local area network* (LAN) provides a proven *broadcast* type of interconnection, which often serves the purpose and is commonly employed.

- We saw that the single processor-memory bus in a shared memory multiprocessor system can become a performance bottleneck, limiting the scalability of such systems.

- In a similar way – i.e. due to contention amongst the *N* processing nodes for access to the shared LAN – does the LAN not also become a performance bottleneck in a computing cluster?

- The answer is: Not necessarily, because:
  - Message *send* and *receive* operations between processes are higher level operations as compared to *read* and *write* operations to main memory.
  - In a running program, each machine instruction may generate an average of, say, 1.4 accesses to main memory.
  - As against this, a message *send* or *receive* operation in a running program may occur once for every 10^5 or even a larger number of machine instructions.
  - Also, application designers have design options to keep message lengths relatively short.
- Therefore the shared high-speed LAN in a computing cluster is less likely to become a performance bottleneck in the system.

Distributed Computing

- Network technology today makes it possible to design a computing application consisting of multiple processes, running in parallel on processing nodes which are inter-connected over large distances.

- Message-passing primitives are employed for communication and synchronization between processes.

- Such a model of computing is known as *distributed computing*, because such a system is not restricted to a single computer system or even a cluster.

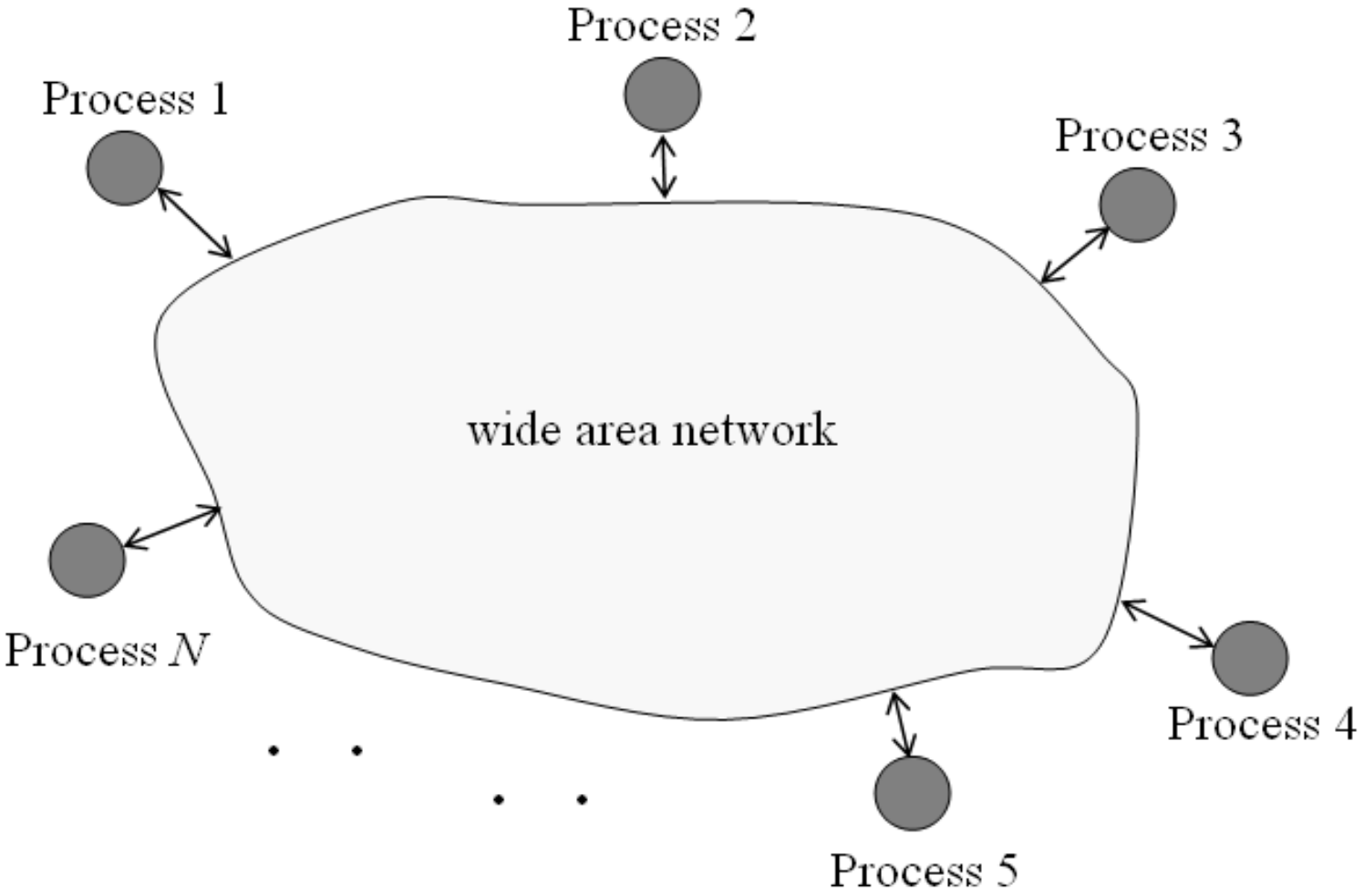
- The computing task is well and truly distributed over multiple systems, and its integrity against network delays is assured by the message-passing model.

- It is natural to model such a distributed computation as a set of processes.

- For this purpose, in fact even details of the computer system available at each computing node of the system are not relevant.

- Recall that in a computing cluster we have assumed the processing nodes to be homogeneous. But that is no longer a constraint in distributed computing.

- Interactions between processes are defined only in terms of *request* and *response* messages exchanged, with no constraint placed on the hardware or software which produces the messages.



- Figure illustrates the concept of distributed computing. Network supports inter-process communication by ensuring reliable message delivery.

- Under this distributed model of computation, we must assume that the single application consists of, say, *N* processes, running on a certain number of computer systems connected to the network.

- The message-passing primitives *send* and *receive* are asynchronous, in the sense that no synchronization is required between sending and receiving processes.

- *Send* and *receive* primitives are built on the basis of reliable delivery of messages by the network – but their functionality is independent of the type and speed of network links, random delays encountered, and other such factors.

- As a result, the model of distributed computing based on message-passing is robust, and is widely used.

- Many of today's web-based applications are in this category.
  - Such web-based architecture usually divides processes into *clients* and *servers*, and sometimes even a *middle layer* between these two.
  - The simplest such architecture has one server and many potential clients, since any node on the web can become a client by *connecting* to the server through the network.
- If aggregate load on the server is heavy, the server may even take the shape of a computing cluster.
- This model of distributed computing is powerful, robust and versatile enough for the enormously vast range of today's web-based applications.

Summary

- Shared memory multiprocessors
- Cache coherence
- Synchronization and arbitration
- Message-passing model of computation
- Cluster computing
- Interconnection structures
- Distributed computing