2.8 Hardware Concepts

Even though all parallel and distributed systems consist of multiple CPUs, there are several different ways the hardware can be organized, especially in terms of *how they are interconnected* and *how they communicate*. Various classification schemes for multiple CPU computer systems have been proposed over the years. A system consisting of multiple interconnected processors can be classified according to its distribution characteristics into two types: ***Tightly coupled*** and ***Loosely coupled.***

(multiprocessing)

1. ***Tightly coupled systems.*** In these systems, there is a single system with primary memory (address space) that is shared by all the processors [Figure 2.9(a)]. If any processor writes, for example, the value 100 to the memory location x, any other processor subsequently reading from location x will get the value 100. Therefore, in these systems, any communication between the processors usually takes place through the shared memory.

multi computers

2. ***Loosely coupled systems.*** In these systems, the processors do not share memory, and each processor has its own local memory [Figure 2.9(b)]. If a processor writes the value 100 to the memory location x, this write operation will only change the contents of its local memory and will not affect the contents of the memory of any other processor. Hence, if another processor reads the memory location x, it will get whatever value was there before in that location of its own local memory. In these systems, all

physical communication between the processors is done by passing messages across the network that interconnects the processors.

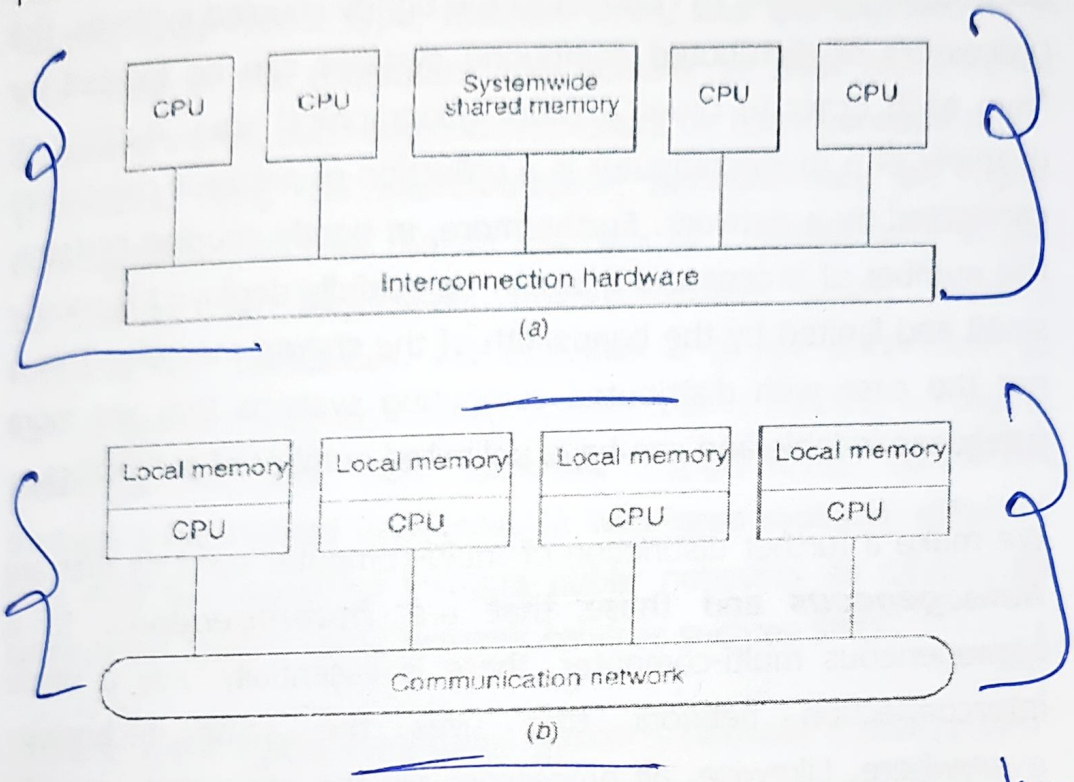

Figure 2.9: tightly coupled and loosely coupled systems:
(a) a tightly coupled multiprocessor system;
(b) a loosely coupled multiprocessor system.

Hence, most computers can be divided into two groups: those that have shared memory, usually called multiprocessors, and those that do not, sometimes called multi-computers. The essential difference is that: in a multiprocessor, there is a single physical address space that is shared by all CPUs. In contrast, in a multi-computer, every machine has its own private memory.

Usually, tightly coupled systems are referred to as parallel processing systems, and loosely coupled systems are referred to as multi-computers, distributed computing systems or simply distributed systems. In contrast to the tightly coupled systems, the processors of distributed computing systems can be located far from each other to cover a wider geographical area. A common example of a multi-computer is a collection of personal computers connected by a network. Furthermore, in tightly coupled systems, the number of processors that can be usefully deployed is usually small and limited by the bandwidth of the shared memory. This is not the case with distributed computing systems that are more freely expandable and can have unlimited number of processors.

We make a further distinction of multi-computer systems that are ***homogeneous*** and those that are ***heterogeneous***. In a homogeneous multi-computer, there is essentially only a single interconnection network that uses the same technology everywhere. Likewise, all processors are the same and generally have access to the same amount of private memory. **Homogeneous multi-computers tend to be used more as parallel systems** (working on a single problem), **just like multiprocessors**. In contrast, a heterogeneous multicomputer system may contain a variety of different, independent computers, which in turn are connected through different networks. For example, a distributed computer system may be constructed from a collection of different local-area computer networks, which are interconnected through an FDDI or ATM-switched backbone.

Generally, **most distributed systems** as they are used today are built on top of a heterogeneous multicomputer. This means that, the computers that form part of the system may vary widely with respect to processor type, memory sizes, and I/O bandwidth. In fact, some of the computers may actually be high performance parallel systems, such as multiprocessors or homogeneous multi-computers. Also, the interconnection network may be highly heterogeneous as well. As an example of heterogeneity is the construction of large-scale multi-computers using existing networks and backbones. For example, it is not uncommon to have a campus-wide distributed system that is running on top of local-area networks from different departments, connected through a high-speed backbone. In wide-area systems, different sites may be connected through public networks as offered by commercial carriers using network services such as SMDS.

In short, a *distributed computing system* is basically a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals, and the communication between any two processors of the system takes place by message passing over the communication network. For a particular processor, its own resources are local, whereas the other processors and their resources are remote. Together, a processor and its resources are usually referred to as a **node or site or machine** of the distributed computing system.

2.9 Software Concepts

Hardware for distributed systems is important, but it is software that largely determines what a distributed system actually looks like. Generally, software of distributed systems is very much like *traditional operating systems*. First, they act as resource managers for the underlying hardware, allowing multiple users and applications to share resources such as CPUs, memories, peripheral devices, the network, and data of all kinds. Second, and perhaps more important, is that distributed systems attempt to hide the intricacies and heterogeneous nature of the underlying hardware by providing a virtual machine on which applications can be easily executed.

To understand the nature of distributed systems, we will first take a look at operating systems in relation to distributed computers. Operating systems for multicomputers can be roughly divided into two categories: ***tightly coupled systems*** and ***loosely-coupled systems***. In tightly-coupled systems, the operating system essentially tries to maintain a single, global view of the resources it manages. Loosely-coupled systems can be thought of as a collection of computers each running their own operating system. However, these operating systems work together to make their own services and resources available to the others. This distinction between tightly-coupled and loosely-coupled operating systems is related to the hardware classification given in the previous section. A ***tightly coupled operating system*** is generally referred to as

a ***Distributed Operating System (DOS)***, and is used for managing multiprocessors and homogeneous multicomputers. Like traditional uniprocessor operating systems, the main goal of a distributed operating system is to hide the intricacies of managing the underlying hardware such that it can be shared by multiple processes.

The ***loosely-coupled Network Operating System (NOS)*** is used for heterogeneous multicomputer systems. Although managing the underlying hardware is an important issue for NOS, the distinction from traditional operating systems comes from the fact local services are made available to remote clients. In the following sections we will first take a look at tightly-coupled and loosely-coupled operating systems.

To actually come to a distributed system, enhancements to the services of network operating systems are needed such that a better support for distribution transparency is provided. These enhancements lead to what is known as ***middleware***, and lie at the heart of modern distributed systems. Middleware is also discussed in this section. Figure 2.10 summarizes the main issues with respect to DOS, NOS, and middleware.

Figure 2.11 shows the 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. Figure 2.11 schematically shows the interaction of this software with these system components at each processor. Here we assume that the middleware layer does not contain the traditional application layer functions of the network protocol stack, such as http, mail, ftp, and telnet.

System	Description	Main goal
DOS	Tightly-coupled operating system for multiprocessors and homogeneous multicomputers	Hide and manage hardware resources
NOS	Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

Figure 2.10: An overview between DOS (Distributed Operating Systems), NOS (Network Operating Systems), and middleware.

Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program's 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. There are several

various classes of multiprocessor/multicomputer operat systems:

- The operating system running on loosely coupled processo (i.e., heterogeneous and/or geographically distant processors which are themselves running loosely coupled software (i.e. software that is heterogeneous), is classified as a networ operating system. In this case, the application cannot run an significant distributed function that is not provided by the application layer of the network protocol stacks on the various processors.
- The operating system running on loosely coupled processors, which are running tightly coupled software (i.e., the middleware software on the processors is homogenous), is classified as a distributed operating system.
- The operating system running on tightly coupled processors, which are themselves running tightly coupled software, is classified as a multiprocessor operating system. Such a parallel system can run sophisticated algorithms contained in the tightly coupled software.

DOS

2.9.1 Distributed Operating Systems

There are two types of distributed operating systems. A ***multiprocessor operating system*** manages the resources of a multiprocessor. A ***multicomputer operating system*** is an operating system that is developed for homogeneous

multicomputers. The functionality of distributed operating systems is essentially the same as that of traditional operating systems for uniprocessor systems, except that they handle multiple CPUs.

2.9.1.1 Multiprocessor Operating Systems

An important, but often not entirely obvious extension to uniprocessor operating systems, is support for multiple processors having access to a shared memory. Conceptually, the extension is simple in that all data structures needed by the operating system to manage the hardware, including the multiple CPUs, are placed into shared memory. The main difference is that these data are now accessible by multiple processors, so that they have to be protected against concurrent access to guarantee consistency.

However, many operating systems, especially those for PCs and workstations, cannot easily handle multiple CPUs. The main reason is that they have been designed as monolithic programs that can be executed only with a single thread of control. Adapting such operating systems for multiprocessors generally means redesigning and reimplementing the entire kernel. Modern operating systems are designed from the start to be able to handle multiple processors.

Multiprocessor operating systems aim to support high performance through multiple CPUs. An important goal is to make the number of CPUs transparent to the application. Achieving such

transparency is relatively easy because the communication between different (parts of) applications uses the same primitives as those in multitasking uniprocessor operating systems. The idea is that all communication is done by manipulating data at shared memory locations, and that we only have to protect that data against simultaneous access. Protection is done through synchronization primitives. Two important (and equivalent) primitives are ***semaphores*** and ***monitors***.

2.9.1.2 Multicomputer Operating Systems

Operating systems for multicomputers are of a totally different structure and complexity than multiprocessor operating systems. This difference is caused by the fact that data structures for systemwide resource management can no longer be easily shared by merely placing them in physically shared memory. Instead, the only means of communication is through message passing. Multicomputer operating systems are therefore generally organized as shown in Figure 2.12.

Each node has its own kernel containing modules for managing local resources such as memory, the local CPU, a local disk, and so on. Also, each node has a separate module for handling interprocessor communication, that is, sending and receiving messages to and from other nodes.

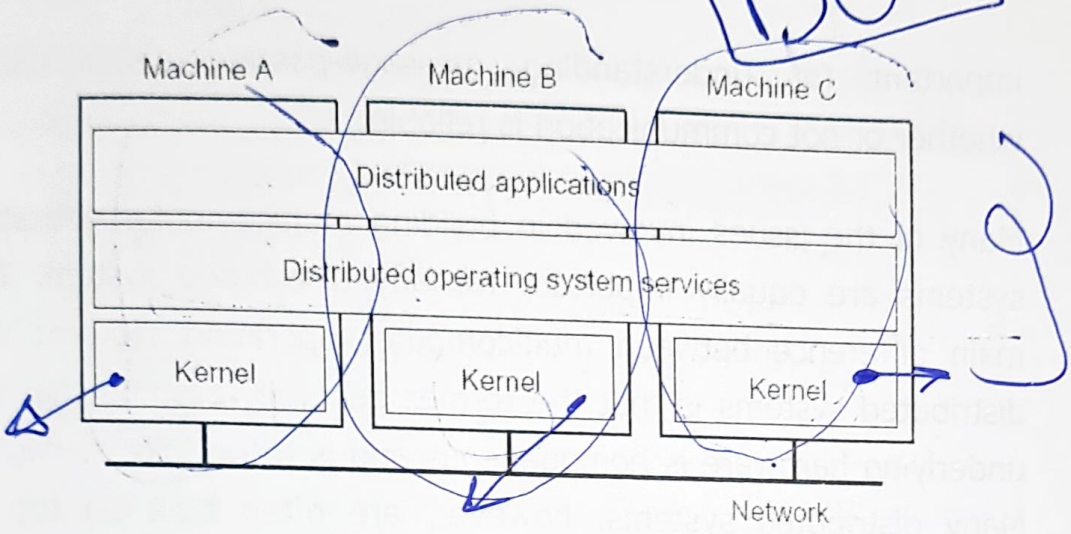

Figure 2.12: General structure of a multicomputer operating system (DOS).

Above each local kernel is a common layer of software that implements the operating system as a virtual machine supporting parallel and concurrent execution of various tasks. In fact, as we shall discuss shortly, this layer may even provide an abstraction of a multiprocessor machine. In other words, it provides a complete software implementation of shared memory. Additional facilities commonly implemented in this layer are, for example, those for assigning a task to a processor, masking hardware failures, providing transparent storage, and general interprocess communication. In other words, facilities that one would normally expect from any operating system. Multicomputer operating systems that do not provide a notion of shared memory can offer only message-passing facilities to applications.

Unfortunately, the semantics of message-passing primitives may vary widely between different systems. Another issue that is

important for understanding message-passing semantics is whether or not communication is reliable.

Many of the issues involved in building multicomputer operating systems are equally important for any distributed system. The main difference between multicomputer operating systems and distributed systems is that the former generally assume that the underlying hardware is homogeneous and is to be fully managed. Many distributed systems, however, are often built on top of existing operating systems, as we will discuss shortly.

each with its own OS

2.9.2 Network Operating Systems

In contrast to distributed operating systems, network operating systems do not assume that the underlying hardware is homogeneous and that it should be managed as if it were a single system. Instead, they are generally constructed from a collection of uniprocessor systems, each with its own operating system, as shown in Fig. 2.13. The machines and their operating systems may be different, but they are all connected to each other in a computer network. Also, network operating systems provide facilities to allow users to make use of the services available on a specific machine. It is perhaps easiest to describe network operating systems by taking a closer look at some services they typically offer.

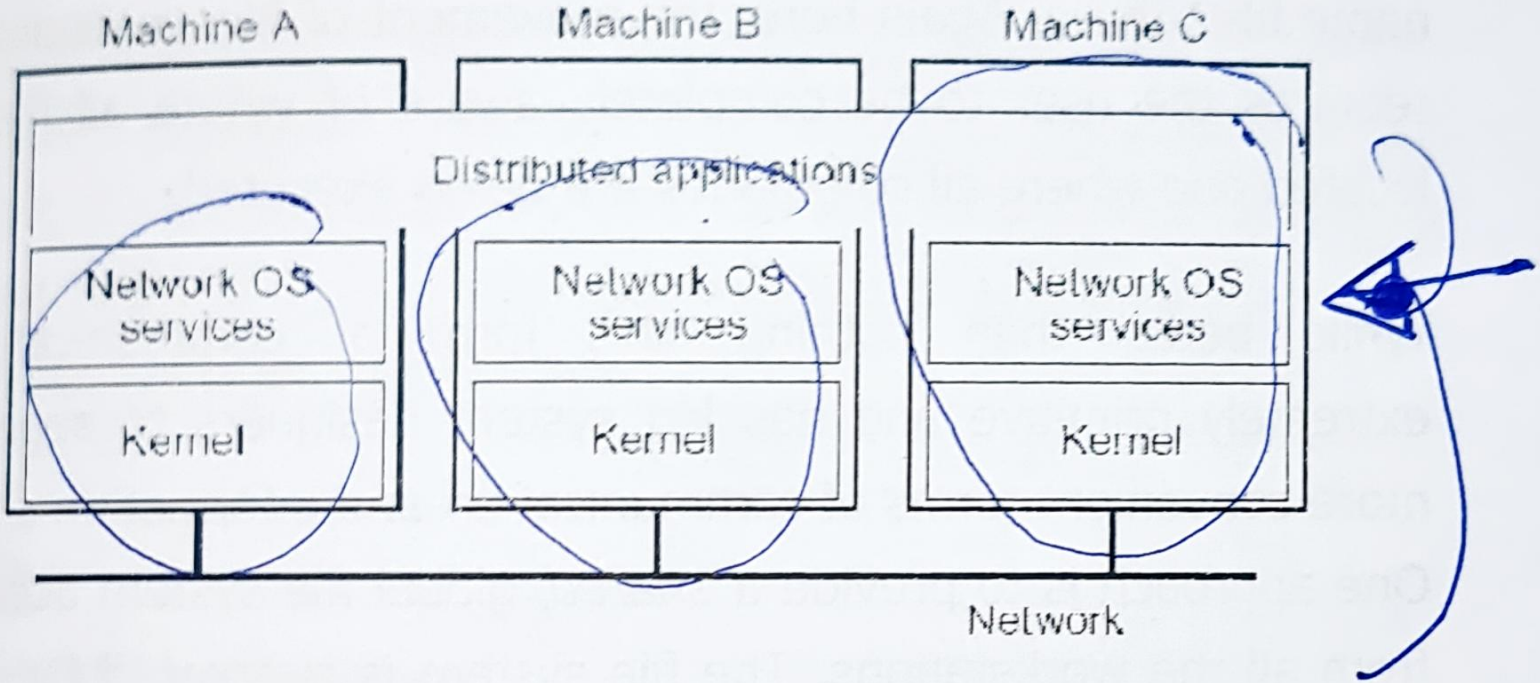

Figure 2.13: General structure of a network operating system (**NOS**).

Network operating systems are clearly more primitive than distributed operating systems. The main distinction between the two types of operating systems is that distributed operating systems make a serious attempt to realize full transparency, that is, provide a single-system view.

2.9.3 Middleware

Neither a distributed operating system nor a network operating system really qualifies as a distributed system according to the above definition. A distributed operating system is not intended to handle a collection of independent computers, while a network operating system does not provide a view of a single coherent system. The question comes to mind whether it is possible to develop a distributed system that has the best of both worlds: the scalability and openness of network operating systems and the transparency and related ease of use of distributed operating systems. The solution is to be found in an additional layer of software that is used in network operating systems to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency. Many modern distributed systems are constructed by means of such an additional layer of what is called ***middleware***. In this section we take a closer look at what middleware actually constitutes by explaining some of its features.

operating system. An important goal is to hide heterogeneity of the underlying platforms from applications. Therefore, many middleware systems offer a more-or-less complete collection of services and discourage using anything else but their interfaces to those services. In other words, skipping the middleware layer and immediately calling services of one of the underlying operating systems is often frowned upon.

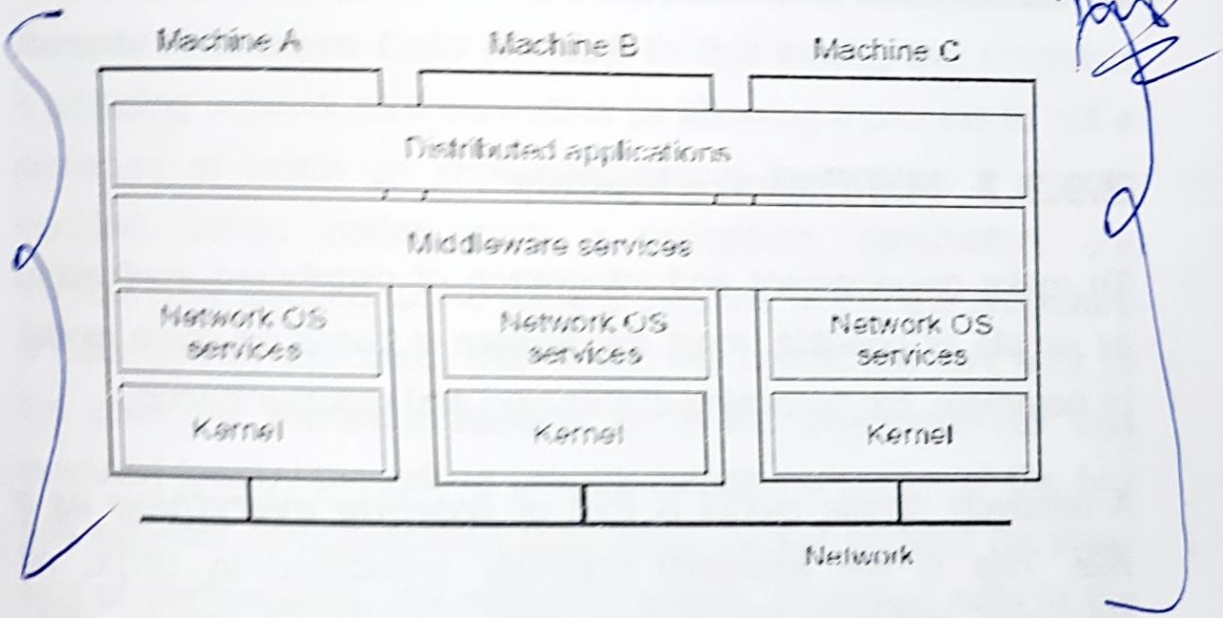

Figure 2.15: A distributed system organized as middleware.

It is interesting to note that middleware was not invented as an academic exercise in achieving distribution transparency. After the introduction and widespread use of network operating systems, many organizations found themselves having lots of networked applications that could not be easily integrated into a single system. At that point, manufacturers started to build higher-level, application-independent services into their systems.

certain nodes due to their specific resource requirements; the required resource(s) may not be available on all the nodes of the system.

4.2.2 Assignment Example

Let us consider the assignment problem of Figure 4.2. It involves only two assignment parameters: the task execution time/cost and the intertask communication time/cost. This system is made up of six tasks $\{t_1, t_2, t_3, t_4, t_5, t_6\}$ and two nodes $\{n_1, n_2\}$. The intertask communication costs/time (C_{ij}) and the execution costs (C_{ip}) of the tasks are given in tabular form in Figure 4.2(a) and 3.2(b), respectively.

An infinite cost for a particular task against a particular node in Figure 4.2(b) indicates that the task cannot be executed on that node due to the task's requirement of specific resources that are not available on that node. Thus, task t_2 cannot be executed on node n_2 and task t_6 cannot be executed on node n_1. In this model of a distributed computing system, there is no parallelism or multitasking of task execution within a program. Thus the total time/cost of process execution consists of the total execution cost of the tasks on their assigned nodes plus the intertask communication costs between tasks assigned to different nodes.

Intertask communications cost						
	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	6	4	0	0	12
t_2	6	0	8	12	3	0
t_3	4	8	0	0	11	0
t_4	0	12	0	0	5	0
t_5	0	3	11	5	0	0
t_6	12	0	0	0	0	0

(a)

Execution costs		
Tasks	Nodes	
	n_1	n_2
t_1	5	10
t_2	2	∞
t_3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

(b)

Serial assignment	
Task	Node
t_1	n_1
t_2	n_1
t_3	n_1
t_4	n_2
t_5	n_2
t_6	n_2

(c)

Optimal assignment	
Task	Node
t_1	n_1
t_2	n_1
t_3	n_1
t_4	n_1
t_5	n_1
t_6	n_2

(d)

Figure 4.2: A task assignment problem example:
(a) intertask communication costs; (b) execution costs of the tasks on the two nodes; (c) serial assignment; (d) optimal assignment.

Figure 4.2(c) shows a serial assignment of the tasks to the two nodes in which the first three tasks are assigned to node n_1, and the remaining three are assigned to node n_2. Observe that this assignment is aimed at ***minimizing the total execution cost.*** But if both the execution costs and the communication costs are taken into account, the total cost

for this assignment comes out to be 58. This cost is calculated as follows:

Serial assignment execution cost (Exec):

$$Exec = X11 + X21 + X31 + X42 + X52 + X62$$
$$= 5 + 2 + 4 + 3 + 2 + 4 = 20.$$

Serial assignment communication cost (Comm):

$$Comm = C14 + C15 + C16 + C24 + C25 + C26 + C34 + C35 + C36$$
$$= 0+0+12+12+3+0+0+11+0 = 38$$

Serial assignment total cost:

$$Cost = Exec + Comm = 20 + 38 = 58$$

Figure 4.2(c) shows an optimal assignment of the tasks to the two nodes that ***minimizes total execution and communication costs.*** In this case, although the execution cost is more than that of the previous assignment, the total assignment cost is only 38. This cost is calculated as follows:

Optimal assignment execution cost (Exec):

$$Exec = X11 + X21 + X31 + X41 + X51 + X62$$
$$= 5 + 2 + 4 + 6 + 5 + 4 = 26$$

Optimal assignment communication cost (Comm):

$$Comm = C16 + C26 + C36 + C46 + C56$$
$$= 12 + 0 + 0 + 0 + 0 = 12$$

Optimal assignment total cost:

$$Cost = Exec + Comm = 26 + 12 = 38$$

Com = C21 + C23 + C25 + C45 + C46 + C56
= 6 + 8 + 3 + 5 = 22

The question now is *how to find the optimal task allocation or assignment.* total time = 28 + 22 = 50

Generally, finding an optimal assignment is a complex problem, hence, several researchers have turned to develop algorithms that are computationally efficient but may yield suboptimal assignments. In the following sections we present the answers to the above question.

4.2.3 Assignment Approaches

A variety of widely differing techniques and methodologies for assigning/scheduling tasks on a distributed system have been proposed. These techniques can be broadly classified into graph theoretical, integer programming and heuristics.

4.2.3.1 Graph Theoretical

The graph theoretical method uses a graph to represent the task assignment problem and then applies the *Minimum-cut Maximum-flow* algorithms on the graph to find an optimal assignment. Using this method, a problem of *M* tasks and *N* processors is modeled as a network in which each processor is a distinguished node and each task is an ordinary node. A