CS 3551 DISTRIBUTED COMPUTING

UNIT I

INTRODUCTION

Introduction: Definition-Relation to Computer System Components – Motivation – Message Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication – Synchronous versus Asynchronous Executions – Design Issues and Challenges; A Model of Distributed Computations: A Distributed Program – A Model of Distributed Executions – Models of Communication Networks – Global State of a Distributed System

Introduction:

1.1 Definition

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

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

- **No common physical clock** - This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.
- **No shared memory -** This is a key feature that requires message-passing for communication. This feature implies the absence of the common physical clock
- **Geographical separation -** The geographically wider apart that the processors are, the more representative is the system of a distributed system.
- **Autonomy and heterogeneity**- The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly.

1.2 Relation to Computer System Components

A typical distributed system is shown in Figure 1.1. Each computer has a memory-processing unit and the computers are connected by a communication network.

Figure 1.2 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.

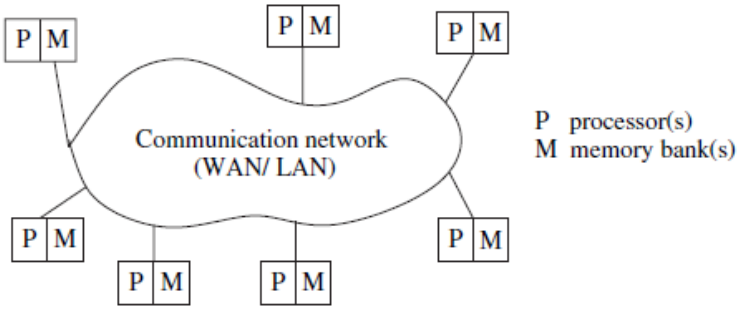

Figure 1.1 A distributed system connects processors by a communication network.

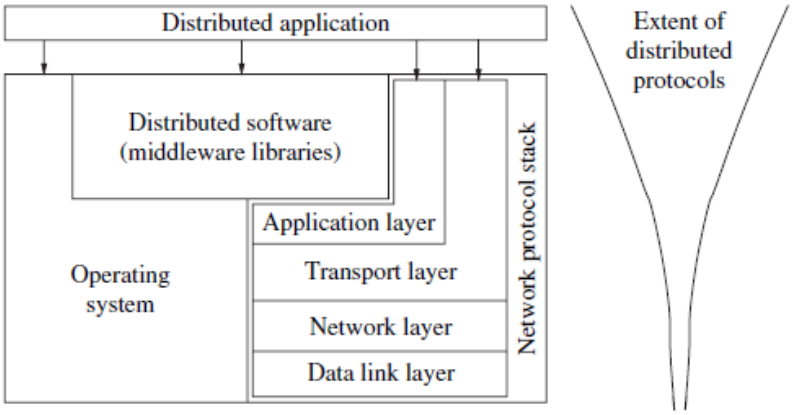

Figure 1.2 Interaction of the software components at each processor.

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 1.2 schematically shows the interaction of this software with these system components at each processor.

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

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

There are several standards such as Object Management Group's (OMG) common object request broker architecture (CORBA), and the remote procedure call (RPC) mechanism.

The RPC mechanism conceptually works like a local procedure call, with the difference that the procedure code may reside on a remote machine, and the RPC software sends a message across the network to invoke the remote procedure. It then awaits a reply, after which the procedure call completes from the perspective of the program that invoked it.

Currently deployed commercial versions of middleware often use CORBA, DCOM (distributed component object model), Java, and RMI (remote method invocation) technologies. The message-passing interface (MPI) developed in the research community is an example of an interface for various communication functions.

1.3 Motivation

The motivation for using a distributed system is some or all of the following requirements.

1. **Inherently distributed computations:** In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.
2. **Resource sharing**: Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective. Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck. Therefore, such resources are typically distributed across the system. For example, distributed databases such as DB2 partition the data sets across several servers, in addition to replicating them at a few sites for rapid access as well as reliability
3. **Access to geographically remote data and resources:** In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated. For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site.

4. **Enhanced reliability:** A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances. Reliability entails several aspects:
    - availability, i.e., the resource should be accessible at all times;
    - integrity, i.e., the value/state of the resource should be correct, in the face of concurrent access from multiple processors, as per the semantics expected by the application;
    - fault-tolerance, i.e., the ability to recover from system failures
5. **Increased performance/cost ratio:** By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased. Although higher throughput has not necessarily been the main objective behind using a distributed system, nevertheless, any task can be partitioned across the various computers in the distributed system.
6. **Scalability**: As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network.
7. **Modularity and incremental expandability:** Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms. Similarly, existing processors may be easily replaced by other processors.

1.4 Message Passing Systems versus Shared Memory Systems

Shared memory systems are those in which there is a (common) shared address space throughout the system. Communication among processors takes place via shared data variables, and control variables for synchronization among the processors. Semaphores and monitors that were originally designed for shared memory uniprocessors and multiprocessors are examples of how synchronization can be achieved in shared memory systems.

All multicomputer (NUMA as well as message-passing) systems that do not have a shared address space provided by the underlying architecture and hardware necessarily communicate by message passing. Conceptually, programmers find it easier to program using shared memory than by message passing.

For this and several other reasons that we examine later, the abstraction called shared memory is sometimes provided to simulate a shared address space. For a distributed system, this abstraction is called distributed shared memory. Implementing this abstraction has a certain cost but it simplifies the task of the application programmer. There also exists a well-known folklore result that communication via message-passing can be simulated by communication via shared memory and vice-versa. Therefore, the two paradigms are equivalent.

Emulating message-passing on a shared memory system (MP →SM)

The shared address space can be partitioned into disjoint parts, one part being assigned to each processor. “Send” and “receive” operations can be implemented by writing to and reading from the destination/sender processor’s address space, respectively.

Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes. A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox. In the simplest case, these mailboxes can be assumed to have unbounded size. The write and read operations need to be controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received.

Emulating shared memory on a message-passing system (SM →MP)

This involves the use of “send” and “receive” operations for “write” and “read” operations. Each shared location can be modeled as a separate process; “write” to a shared location is emulated by sending an update message to the corresponding owner process; a “read” to a shared location is emulated by sending a query message to the owner process. As accessing another processor’s memory requires send and receive operations, this emulation is expensive.

Thus, the latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication under the covers.

In a MIMD message-passing multicomputer system, each “processor” may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory. Between two computers, the communication is by message passing. As message-passing systems are more common and more suited for wide-area distributed systems, we will consider message-passing systems more extensively than we consider shared memory systems.