CATHOLIC UNIVERSITY OF RWANDA

P.O. Box 49 BUTARE / HUYE - RWANDA
TEL: (00250) 0252 530 893 FAX: (00250) 0252 530 627
E-mail: administration@cur.ac.rw
Web: www.cur.ac.rw

FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

LEVEL V WEEKEND PROGRAM

MODULE TITLE: PARALLEL AND DISTRIBUTED SYSTEM (PDSY5513) SYLLABUS

LECTURER: Mr. UWIMANA Jean Pierre

ACADEMIC YEAR: 2016-2017

Table of Contents

PART ONE: PARALLEL SYSTEM

Goals of course

- Understand architecture of modern parallel systems.
- Employ software technologies for parallel programming.
- Design efficient and two-fold generic parallel solutions.
- For a wide variety of parallel systems & broad class of similar algorithms.
- Sharpen your low-level and high-level IT skills.
- Understand their performance.
- Make successful technology**.**

CHAPTER 0.INTRODUCTION

What is a system?

System is a group of things worked together for common goal or even we can say process of things that happen together.

What are Computer systems?

Computer systems include the computer along with any software and peripheral devices that are necessary to make the computer function. Every computer system, for example, requires an operating system.

System Boundaries

System boundaries being independent entity the system of any kind has its boundaries (limits) that set and separate it apart from other external entities within its interests is the system boundaries.

WHAT IS MICROPROCESSOR

The microprocessor is one type of ultra-large-scale integrated circuit. Integrated circuits, also known as microchips or chips, are complex electronic circuits consisting of extremely tiny components

formed on a single, thin, flat piece of material known as a semiconductor. Modern microprocessors incorporate transistors (which act as electronic amplifiers, oscillators, or, most commonly, switches), in addition to other components such as resistors, diodes, capacitors, and wires, all packed into an area about the size of a postage stamp.

A microprocessor consists of several different sections:
The arithmetic/logic unit (ALU) performs calculations on numbers and makes logical decisions.
The registers are special memory locations for storing temporary information much as a scratch pad does.
The control unit deciphers programs; buses carry digital information throughout the chip and computer; and local memory supports on-chip computation.
More complex microprocessors often contain other sections such as sections of specialized memory, called **cache memory**, to speed up access to external data-storage devices. Modern microprocessors operate with bus *widths* of 64 *bits* (binary digits, or units of information represented as 1s and 0s), meaning that 64 bits of data can be transferred at the same time.

A crystal oscillator in the computer provides a clock signal to coordinate all activities of the microprocessor. The clock speed of the most advanced microprocessors allows billions of computer instructions to be executed every second

Figure 0.1. **Microprocessor**

Construction of microprocessor

Microprocessors are fabricated using techniques similar to those used for other integrated circuits, such as memory chips. Microprocessors generally have a more complex structure than do other chips, and their manufacture requires extremely precise techniques.

Economical manufacturing of microprocessors requires mass production. Several hundred *dies*, or circuit patterns, are created on the surface of a silicon wafer simultaneously. Microprocessors are constructed by a process of deposition and removal of conducting, insulating, and semiconducting materials one thin layer at a time until, after hundreds of separate steps, a complex sandwich is constructed that contains all the interconnected circuitry of the microprocessor. Only the outer surface of the silicon wafer a layer about 10 microns (about 0.01 mm/0.0004 in) thick or about one-tenth the thickness of a human hair is used for the electronic circuit. The processing steps include substrate creation, oxidation, lithography, etching, ion implantation, and film deposition.

The first step in producing a microprocessor is the creation of an ultrapure silicon substrate, a silicon slice in the shape of a round wafer that is polished to a mirror-like smoothness. At present, the largest wafers used in industry are 300 mm (12 in) in diameter.

In the oxidation step, an electrically no conducting layer, called a dielectric, is placed between each conductive layer on the wafer. The most important type of dielectric is silicon dioxide, which is "grown" by exposing the silicon wafer to oxygen in a furnace at about 1000°C (about 1800°F). The oxygen combines with the silicon to form a thin layer of oxide about 75 angstroms deep (an angstrom is one ten-billionth of a meter).

Nearly every layer that is deposited on the wafer must be patterned accurately into the shape of the transistors and other electronic elements. Usually this is done in a process known as photolithography, which is analogous to transforming the wafer into a piece of photographic film and projecting a picture of the circuit on it. A coating on the surface of the wafer, called the photoresist or resist, changes when exposed to light, making it easy to dissolve in a developing solution.

These patterns are as small as 0.13 microns in size. Because the shortest wavelength of visible light is about 0.5 microns, short-wavelength ultraviolet light must be used to resolve the tiny details of the

patterns. After photolithography, the wafer is etched that is, the resist is removed from the wafer either by chemicals, in a process known as wet etching, or by exposure to a corrosive gas, called a plasma, in a special vacuum chamber.

In the next step of the process, ion implantation, also called doping, impurities such as boron and phosphorus are introduced into the silicon to alter its conductivity. This is accomplished by ionizing the boron or phosphorus atoms (stripping off one or two electrons) and propelling them at the wafer with an ion implanter at very high energies. The ions become embedded in the surface of the wafer.

The thin layers used to build up a microprocessor are referred to as films. In the final step of the process, the films are deposited using sputterers in which thin films are grown in plasma; by means of evaporation, whereby the material is melted and then evaporated coating the wafer; or by means of chemical-vapor deposition, whereby the material condenses from a gas at low or atmospheric pressure. In each case, the film must be of high purity and its thickness must be controlled within a small fraction of a micron.

Microprocessor architecture

There is four major functional element of a microprocessor system, namely: the control unit, Arithmetic logic unit, Registers and bus, as shown in the figure below.

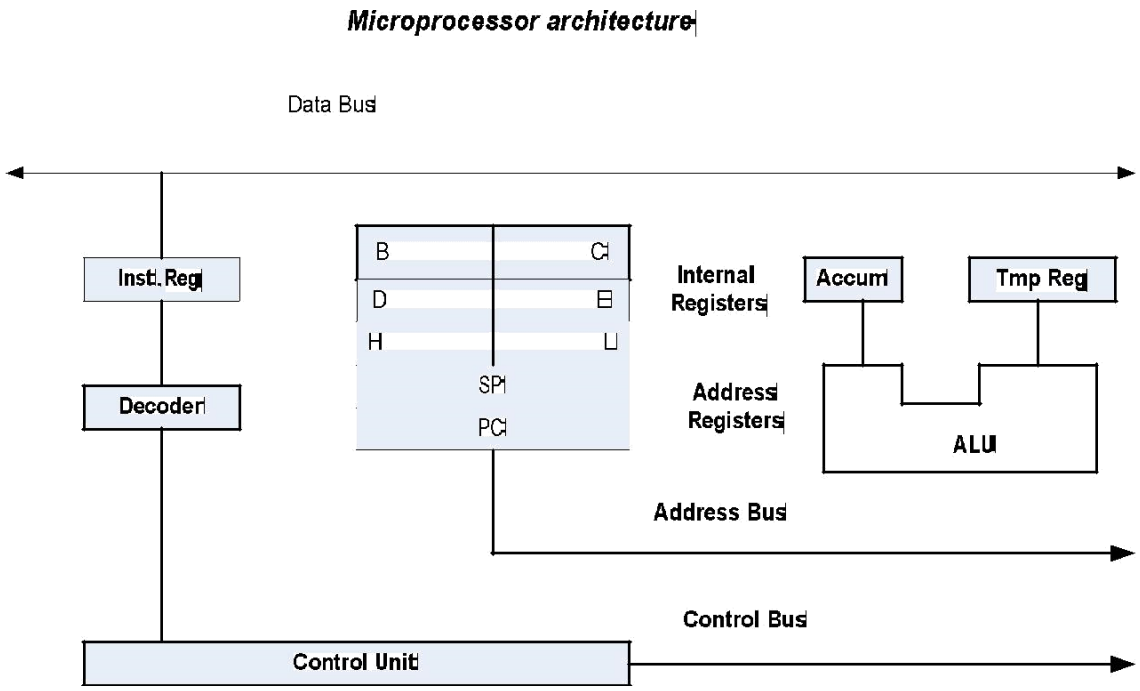


Figure 0.2.Microprocessor architecture

a. Control unit

The control unit (brain) controls the entire operation of the microprocessor system. The control unit generates synchronization signals and manages commands exchanged between the ALU, I/O and the memory. The control unit fetches, decodes and executes instructions.

b. Arithmetic Logic Unit

The ALU performs arithmetic and logical operations (the basic data transformation in a microprocessor). The Accumulator register holds one operand, while temporary register holds the other operand. The result is usually stored in accumulator.

c. Internal registers

Internal registers are fast memories with the microprocessor used as temporary data holding areas to enable ALU to manipulate data at high speed. In original microprocessors, the internal registers B, C, D and E were used to store 8-bit data.

- **Address Registers.**

There are 16-bit or more registers for the storage of addresses. They are connected to the address bus. They consist of the program counter, the stack pointer and the Index Register (H and L).

- **Program counters (PC).**

The program counter contains the address of next instruction to be executed. To execute the next instruction, it must be brought from the memory into the microprocessor.

- **Stack Pointer**

Stack pointer contains the address of the stack within the memory. The stack contains addresses of switching tasks, subroutines, etc.

- **Index Register**

Indexing is used to access blocks of data in the memory using a single instruction. An index register contains either a displacement that is automatically added to a base, or a base that is added to a displacement.

- **Computer bus**

A computer bus consists of a number of wires connecting a processor to another device.

There are three types of bus and the control bus. The microprocessor communicates with the external memory and all I/O devices via the buses.

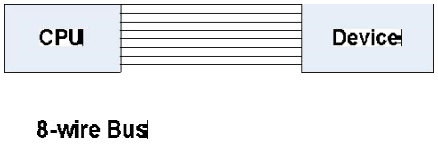


Figure 0.5.computer bus

The Microprocessor Operation

During normal operation, the microprocessor sequentially fetches and executes instructions. Each instruction is executed as a sequence of three phases: Fetch, Decode and Execute. The operation cycle is as illustrated below.

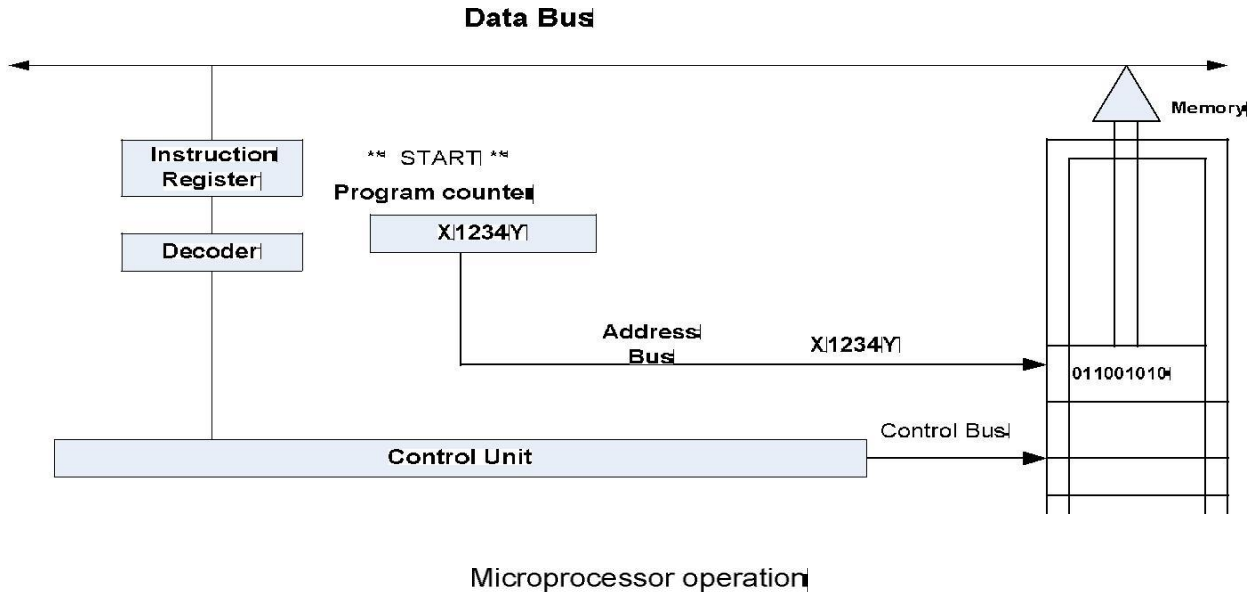


Figure 0.6.Microprocessor operation

Fetch phase

The address of the instruction to be executed, loaded in the Program Counter from a computer program, is used to locate the instruction in the memory or I/O device. Theinstruction is then fetched from the memory or I/O device via the Data bus into the Instruction Register.

Decoding phase

Once the instruction is in the Instruction Register, it is decoded by the Decoder of the Control Unit. The decoding is done to enable the microprocessor determine the kind of operation it is required to perform. Decoding is only performed on instruction and not data. The instruction and data in a typical command such as "ADD 4 to 7"is as follows

ADD = instruction

4 and 7 = data

Execution

The control unit issues appropriate sequence of signals depending on the decoded information.

Typical Operational Procedure

Suppose a microprocessor is to add 10000001 to 11100111.First, for these two numbers to be added they must be stored in main memory.

The ADD instruction is fetched and placed in the Instruction Register where it is decoded.

The element 10000001 is then fetched and placed in one of microprocessors Internal Registers (say register B).

Next, the data 11100111 is fetched and also placed in one of the Internal Registers, and then the ALU is used to perform the additional operation.

Classification of microprocessor

The basic parameters are used to classify microprocessors: Speed and Data Bus Width.

a. Speed

Every microprocessor has a clock that drives its operations. Microprocessors with faster clocks perform operations much faster compared to those with slower clocks. To illustrate how the speed

of microprocessor clocks affect the speed of their operation, suppose that a slow and fast microprocessors are to ADD 4 to 7.

b.Bus width

The bus width of a microprocessor is determined by its Data Bus width .Higher bus widths provide higher performance computationally. For example to fetch a 16-bit instruction from memory using a data bus with of 16-bits would require a single fetch operation, whereas an 8-bit Data bus would require two fetch cycles to fetch the same instruction, thereby slowing the execution of instruction

Microprocessor Timing

ADD 4 to 7 is typical instruction or command that would normally be given to a computer to execute. It requires that the ADD instruction as well as the data 4 and 7 to be added should be fetched.

Advanced CPUs

The microprocessor family described in the last section is all manufactured by Intel Inc, specifically for Microcomputer. Other advanced CPUs exist that are used for manufacturing Minicomputers. These CPUs are usually referred to as RISC (Reduced Instruction set Computers) processors. By design, RISC machines are a lot faster than Intel processor based machine.

CHAPTER I. PARALLEL COMPUTING

Most computers having only one CPU. However, there is a trend towards multiprocessor systems; such systems have more than one processor in close communication, sharing the computer bus, the clock, and sometimes memory and peripheral devices. These systems are referred as "**Tightly coupled**" systems. A system consisting of more than one processor and it is a tightly coupled, and then the system is Parallel system. In parallel systems number of processors executing their jobs parallel. There are several reasons for buildings such systems.

Definition

A system is said to be a Parallel System in which multiple processor have direct access to shared memory which forms a common address space. Usually tightly-coupled system are referred to as Parallel System. In these systems, there is a single system wide primary memory (address space) that is shared by all the processors. On the other hand Distributed System are loosely-coupled system. Parallel computing is the use of two or more processors (cores, computers) in combination to solve a single problem. Parallel machines are becoming quite common and affordable prices of microprocessors, memory and disks have dropped sharply recent desktop computers feature multiple processors and this trend is projected to accelerate.

Applications of Parallel System

An example of Parallel computing would be two servers that share the workload of routing mail, managing connections to an accounting system or database, solving a mathematical problem etc Supercomputers are usually placed in parallel system architecture Terminals connected to single server Advantages of Parallel System Provide Concurrency(do multiple things at the same time) Taking advantage of non-local resources Cost Savings Overcoming memory constraints Save time and money Global address space provides a user-friendly programming perspective to memory .

Parallelism is a digital computer performing more than one task at the same time

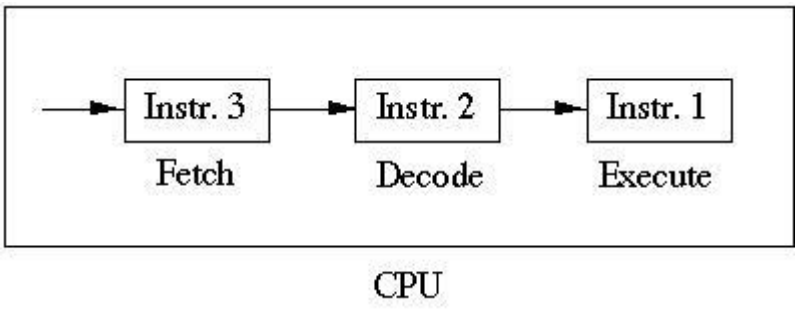
Examples

IO chips

Most computers contain special circuits for IO devices which allow some task to be performed in parallel

Pipelining of Instructions

Some CPU's pipeline the execution of instructions



Multiple Arithmetic units (AU)

Some CPUs contain multiple AU so it can perform more than one arithmetic operation at the same time

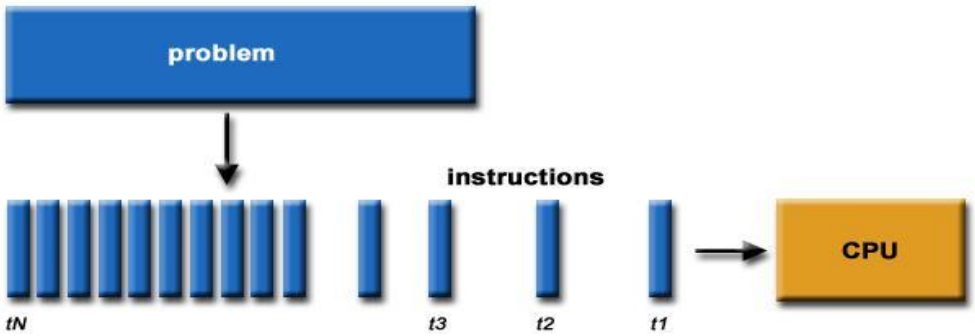
Disadvantages of Parallel System

Primary disadvantage is the lack of scalability between memory and CPUs. Programmer responsibility for synchronization constructs that ensure "correct" access of global memory. It becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

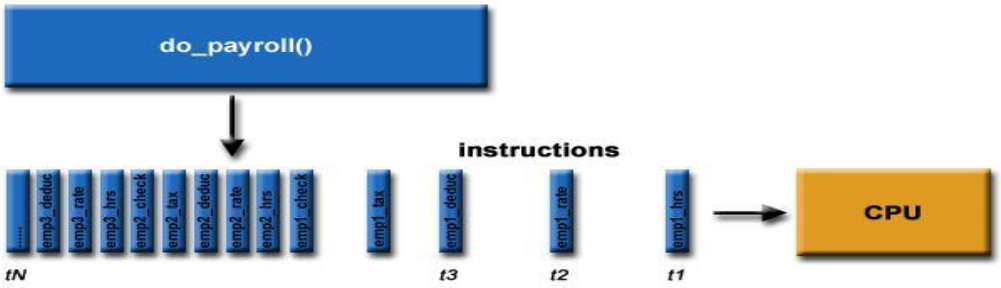
Parallel computation vs. serial computation

Traditionally, software has been written for **serial computation**: To be run on a single computer having a single Central Processing Unit (CPU); A problem is broken into a discrete series of instructions. Instructions are executed one after another. Only one instruction may execute at any moment in time.

Figure6: serial computation



Example



In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem: To be run using multiple processors.

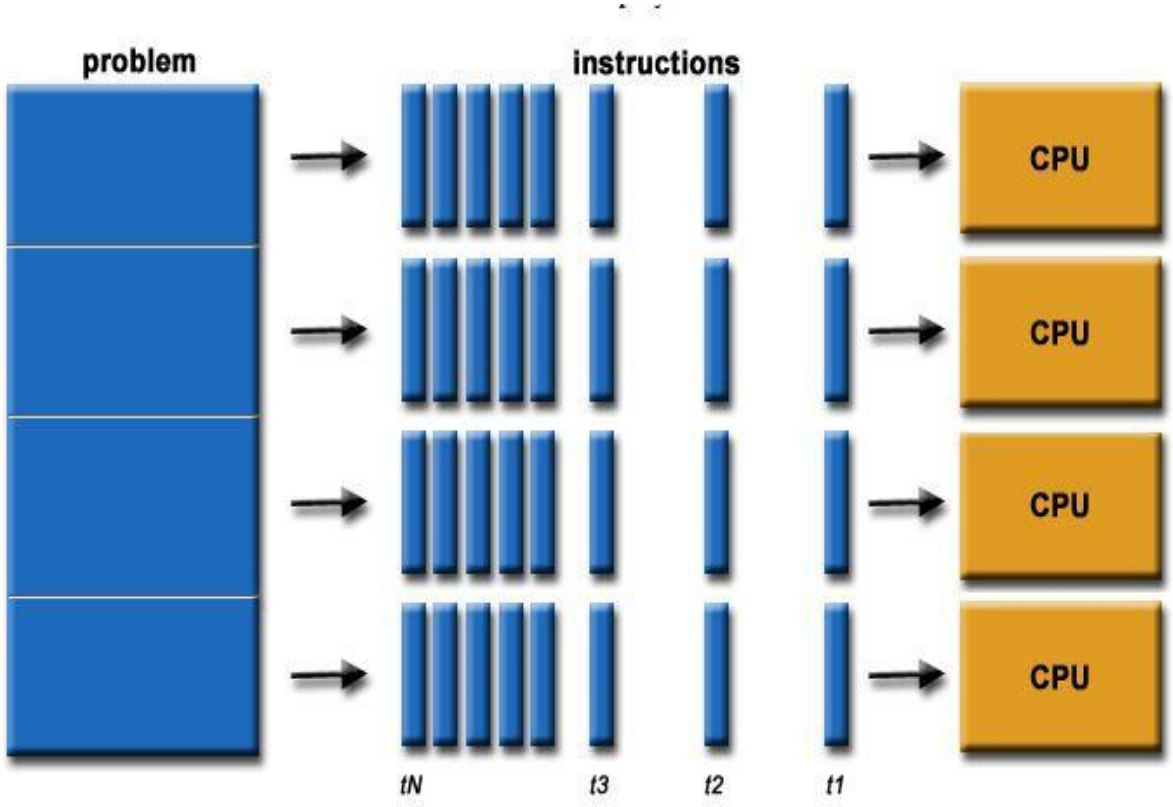
A problem is broken into discrete parts that can be solved concurrently

Each part is further broken down to a series of instructions

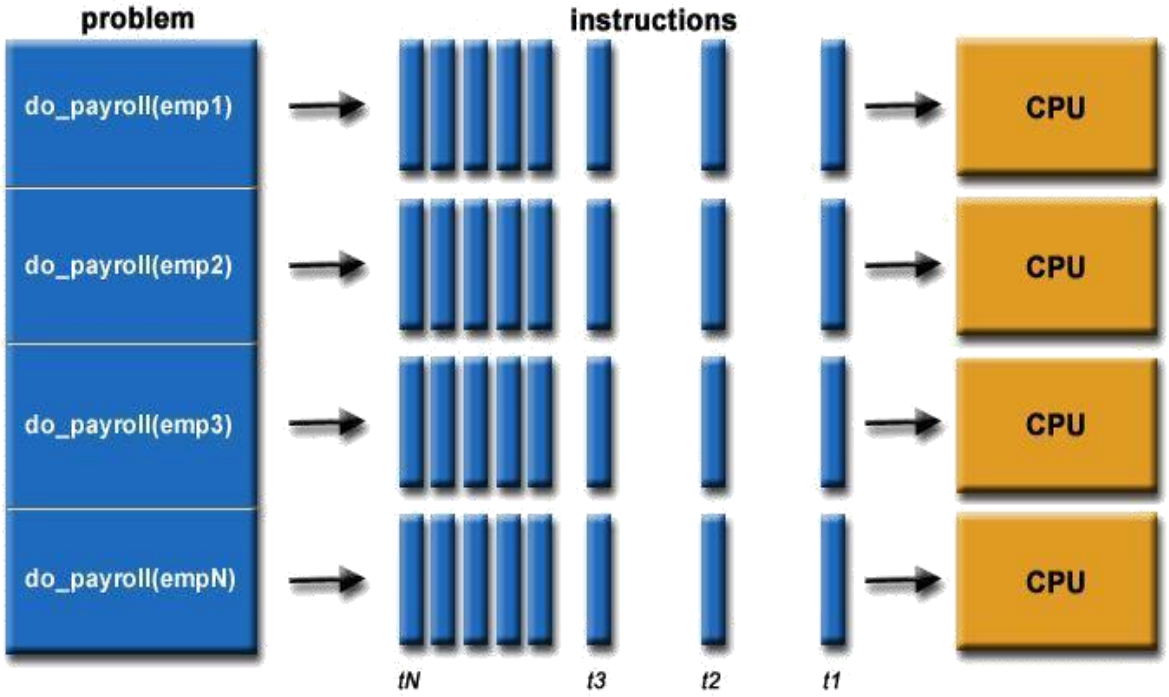
Instructions from each part execute simultaneously on different processors

An overall control/coordination mechanism is employed.

Figure7: Parallel computation



For example:



The computational problem should be able to:

- Be broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Be solved in less time with multiple compute resources than with a single compute resource.

The computer resources might be:

- A single computer with multiple processors
- An arbitrary number of computers connected by a network
- A combination of both

I.1 PARALLEL PROCESSING

Parallel Processing is computer technique in which multiple operations are carried out simultaneously. Parallelism reduces computational time. For this reason, it is used for many computationally intensive applications such as predicting economic trends or generating visual special effects for feature films.

Two common ways that parallel processing is accomplished are through **multiprocessing** or **instruction-level parallelism**.

Multiprocessing links several processors or microprocessors (the electronic circuits that provide the computational power and control of computers) together to solve a single problem.
Instruction-level parallelism uses a single computer processor that executes multiple instructions simultaneously. If a problem is divided evenly into ten independent parts that are solved simultaneously on ten computers, then the solution requires one tenth of the time it would take on a single nonparallel computer where each part is solved in sequential order.

I.2.PARALLEL ARCHITECTURE

In 1966 American electrical engineer Michael Flynn distinguished four classes of processor architecture (the design of how processors manipulate data and instructions). Data can be sent either to a computer's processor one at a time, in **a single data stream**, or several pieces of data can be sent at the same time, in **multiple data streams**. Similarly, instructions can be carried out either one at a time, in a **single instruction stream**, or several instructions can be carried out simultaneously, in **multiple instruction streams.**

Single Instruction stream, Single Data stream (SISD): One piece of data is sent to one processor. For example, if 100 numbers had to be multiplied by the number 3, each number would be sent to the processor, multiplied, and the result stored; then the next number would be sent and calculated, until all 100 results were calculated. Applications that are suited for SISD architectures include those that require complex interdependent decisions, such as word processing.

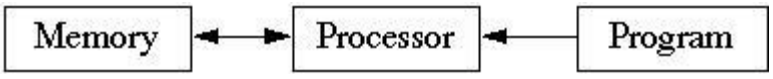
Multiple Instruction stream, Single Data stream (MISD): Processor replicates a stream of data and sends it to multiple processors, each of which then executes a separate program.

For example, the contents of a database could be sent simultaneously to several processors, each of which would search for a different value. Problems well-suited to MISD parallel processing include computer vision systems that extract multiple features, such as vegetation, geological features, or manufactured objects, from a single satellite image.

Single Instruction stream, Multiple Data stream (SIMD): Multiple processing elements that carry out the same instruction on separate data. For example, a SIMD machine with 100 processing elements can simultaneously multiply 100 numbers each by the number 3. SIMD processors are programmed much like SISD processors, but their operations occur on arrays of data instead of individual values. SIMD processors are therefore also known as array processors. Examples of applications that use SIMD architecture are image-enhancement processing and radar processing for air-traffic control.

Multiple Instruction stream, Multiple Data stream (MIMD): Processor has separate instructions for each stream of data. This architecture is the most flexible, but it is also the most difficult to program because it requires additional instructions to coordinate the actions of the processors. It also can simulate any of the other architectures but with less efficiency. MIMD designs are used on complex simulations, such as projecting city growth and development patterns, and in some artificial-intelligence programs.

SISD - Single Instruction Single Data. Sequential Computer

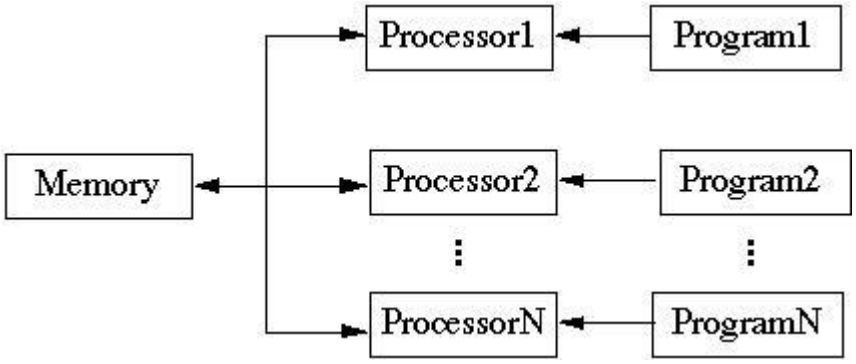


MISD - Multiple Instruction Single Data

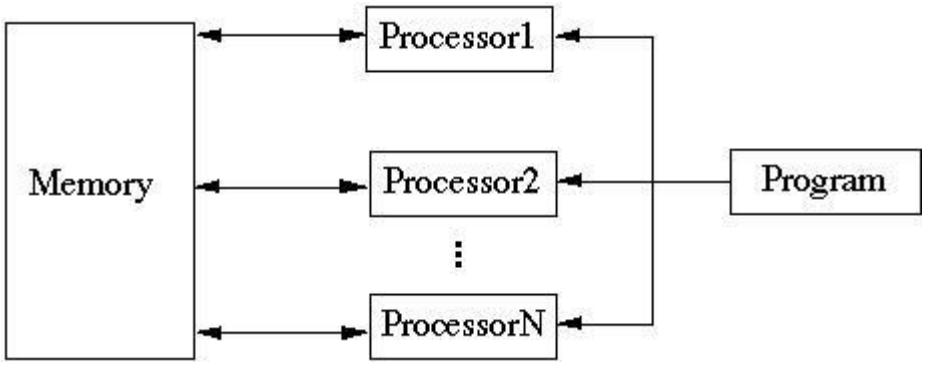
Each processor can do different things to the same input

Example: Detect shapes in an image.

Each processor searches for a different shape in the input image



SIMD - Single Instruction Multiple Data, Each processor does the same thing to different data, Requires global synchronization mechanism ,Each processor knows its id number ,Not all shared memory computers are SIMD



MIMD(Multiple Instruction Multiple Data): Each processor can run different programs on different data MIMD can be shared memory or message passing Can simulate SIMD or SISD if there is global synchronization mechanism Communication is the main issue Harder to program than SIMD

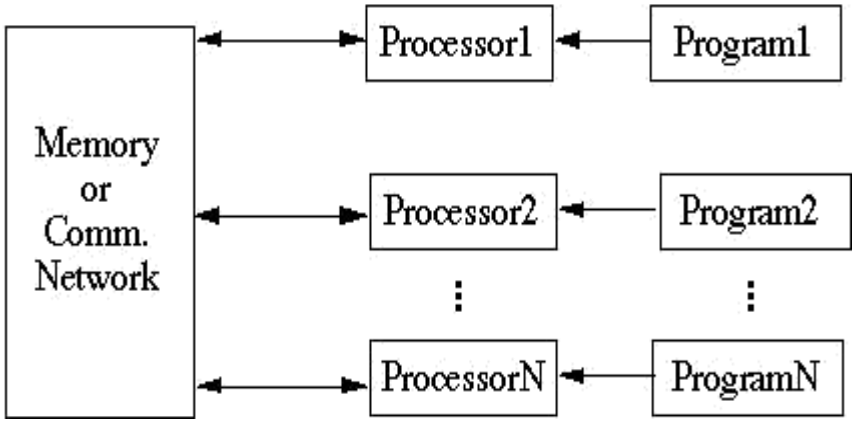
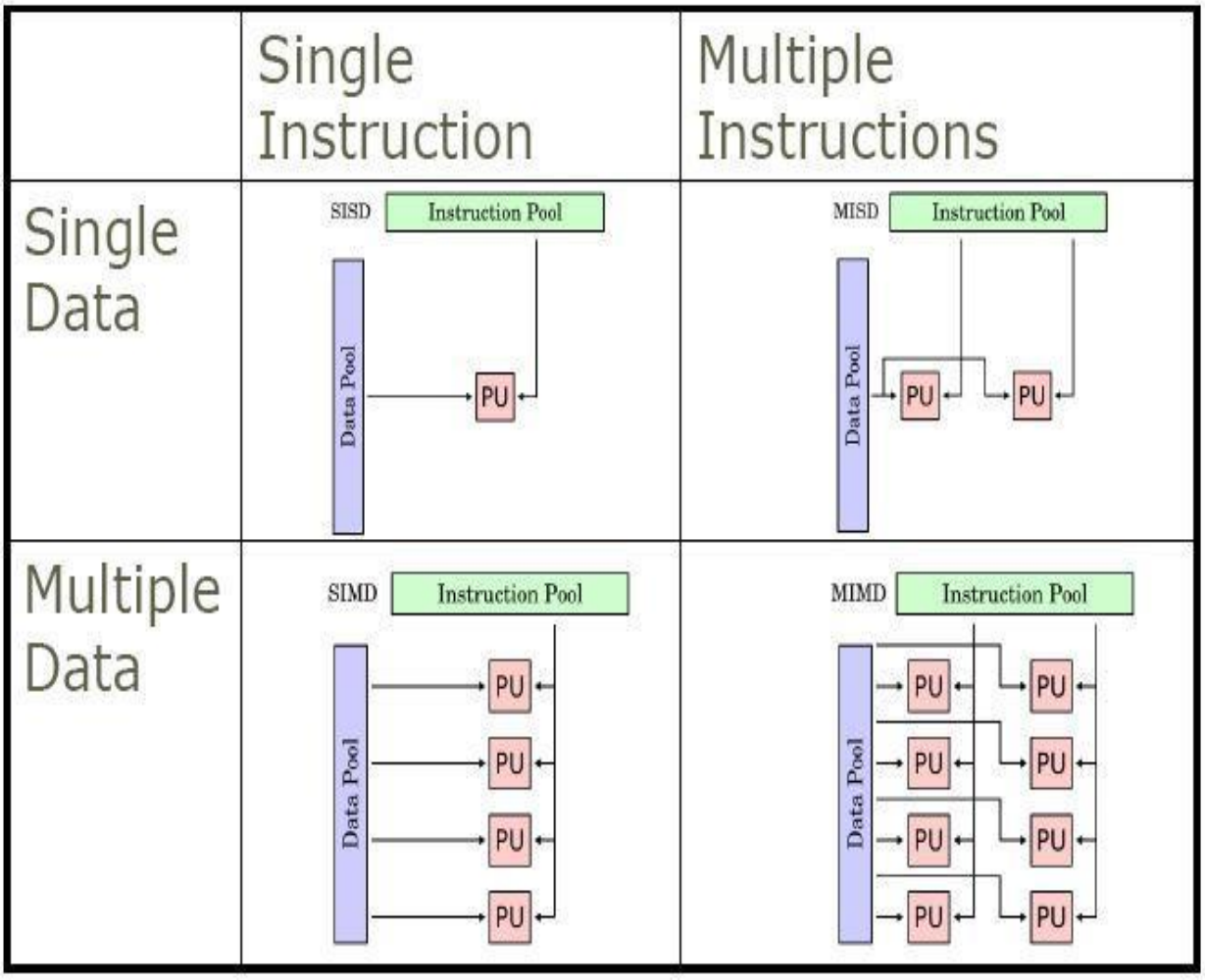


Figure3 : Parallel architecture



I.3.PARALLEL COMMUNICATION

Another factor in parallel-processing architecture is how processors communicate with each other. One approach is to let processors share a single memory and communicate by reading each other's data. This is called **shared memory**. In this architecture, all the data can be accessed by any processor, but care must be taken to prevent the linked processors from inadvertently overwriting each other's results.

An alternative method is to connect the processors and allow them to send messages to each other. This technique is known as **message passing or distributed memory**. Data are divided and stored in the memories of different processors. This makes it difficult to share information because the processors are not connected to the same memory, but it is also safer because the results cannot be overwritten.

In shared memory systems, as the number of processors increases, access to the single memory becomes difficult, and a bottleneck forms. To address this limitation the problem of isolated memory in distributed memory systems, distributed memory processors also can be constructed with circuitry that allows different processors to access each other's memory. This hybrid approach, known as distributed shared memory, eliminates the bottleneck and sharing problems of both architectures.

Message-passing

Processors can only access their own memory and communicate through messages (two-sided mechanism).

Requires the least hardware support.

Easier to debug.

- Interactions happens in well-defined program parts
- The process is in control of its memory!
- Cumbersome communication protocol is needed
- Remote data cannot be accessed directly, only via request.

Instruction-Level Parallelism (ILP): The lifecycle of an instruction

1. Fetch the next instruction from the address stored in the program counter.

2. Store that instruction in the instruction register and **decode** it, and increment the address in the program counter.

3. Execute the instruction currently in the instruction register. If the instruction is not a branch instruction but an arithmetic instruction, send it to the proper ALU.

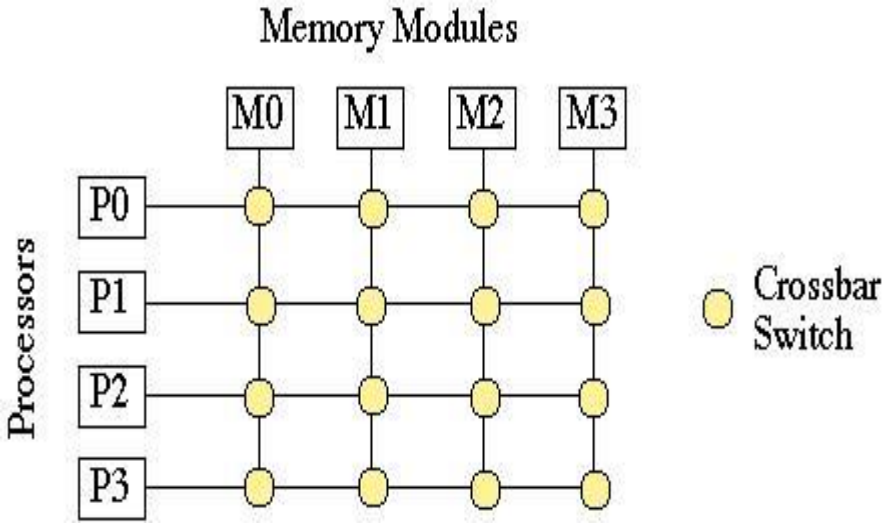
a. Read the contents of the input registers. b. Add

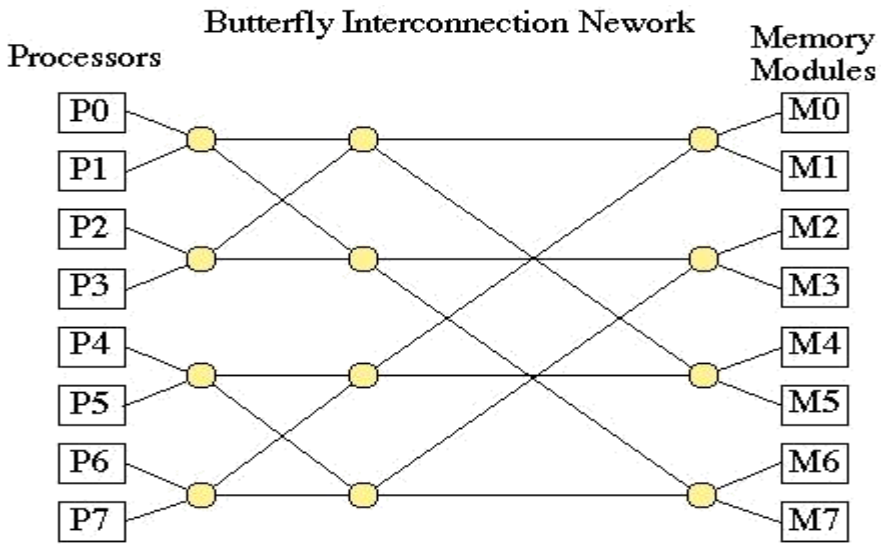
the contents of the input registers.

4. **Write** the results of that instruction from the ALU back into the destination register.

Processor-Memory Interconnection Network

Multiprocessor

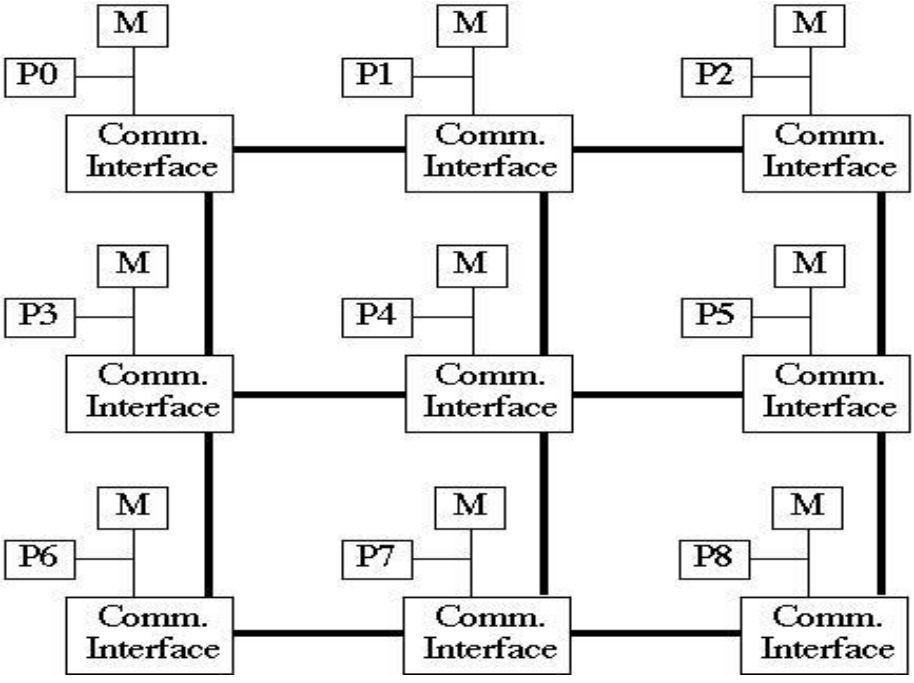




Message Passing or Multicomputers.

Individual processors local memory; Processors communicate via a communication network.

Mesh Communication Network



I.4.COST OF PARALLEL COMPUTING

Parallel processing is more costly than serial computing because multiple processors are expensive and the speedup in computation is rarely proportional to the number of additional processors. MIMD processors require complex programming to coordinate their actions. Finding MIMD programming errors also is complicated by time-dependent interactions between processors.

For example, one processor might require the result from a second processor's memory before that processor has produced the result and put it into its memory. This results in an error that is difficult to identify. Programs written for one parallel architecture seldom run efficiently on another. As a result, to use one program on two different parallel processors often involves a costly and time-consuming rewrite of that program.

CHAPTER II.CONCURRENCY AND CONCURRENCY CONTROL

Concurrency means to do multiple things at once (parallelism, multitasking). But concurrency introduces nondeterminism: the exact order in which things are done (the schedule) is not known in advance. Nondeterminism occurs because the schedule results from the interaction of a system and its scheduling policies with various asynchronous external processes, including physical processes, whose timings are not known or controlled by the system.

It is often the case that some schedules will lead to correct outcomes and some schedules will not. Thus it becomes necessary for programmers to express constraints to prevent the system from selecting or allowing a schedule that yields an incorrect outcome. This is *concurrency control*. Understanding concurrency is a difficult intellectual challenge. Controlling concurrency is an art and a craft. It is important to understand the problem of concurrency and the tools of the art of concurrent programming.

In earlier days concurrency control was something that operating system kernel designers worried about. To the extent that concurrency was a property of the hardware it was up to the kernel to control it. The process model of early operating systems (single-threaded processes with strong isolation interacting through kernel abstractions such as pipes, files, and process fork/wait) limited concurrent interactions to the kernel.

Today concurrency has leaked into application programs. Rich GUIs must respond to user events while other activities are in progress. Concurrency is inherent to distributed applications, which must process messages arriving asynchronously from peers. In particular, high-throughput servers must process concurrent requests from many clients.

Abstractions for concurrent programming

The problem of concurrency control is fundamental and independent of the various programming models that exist to express or provide concurrency: events, threads, processes. There is a perennial disagreement about which abstractions are easiest for programmers to use in writing correct concurrent programs. For example, the recent interest in the MapReduce abstraction for cluster computing ("cloud computing") is based in part on a belief that it enables even naive programmers to write correct parallel programs that scale to very large clusters. The problem of safe concurrent programming is even more

important with the prevalence of multicore processors.

Environments for safe concurrent programming generally create higher-level constructs that minimize direct sharing of data structures in memory by multiple threads. Instead they pass data among threads using messages or data streams. For example, an event-driven program might consist of single-threaded objects that receive streams of events. The systems research community generates a steady stream of papers exploring the relative merits of various models for safe concurrent programming model from the standpoint of ease of use and performance.

However, these higher-level models use threads under the hood. And threads are still the most commonly encountered model for concurrent programming. The best way to understand concurrency is to study threads and related abstractions (synchronization primitives) for controlling concurrency in multi-threaded programs.

Concurrent and parallel execution

Concurrency and parallelism are two related but distinct concepts.

Concurrency means, essentially, that task A and task B both need to happen independently of each other, and A starts running, and then B starts before A is finished.

There are various different ways of accomplishing concurrency. One of them is parallelism--having multiple CPUs working on the different tasks at the same time. But that's not the only way. Another is by *task switching*, which works like this: Task A works up to a certain point, then the CPU working on it stops and switches over to task B, works on it for a while, and then switches back to task A. If the time slices are small enough, it may appear to the user that both things are being run in parallel, even though they're actually being processed in serial by a multitasking CPU.

CHAPTER III. PROCESSES AND THREADS SYNCHRONIZATION

Multithreading

A process is an instance of the program in execution. One process is split into separate threads, executing a different sequence of instructions and having access to the same memory.

Synchronization is cooperating threads, not a single operation. Since processes in concurrent systems frequently need to communicate with other processes therefore, there is a need for a well structured communication, without using interrupts, among processes.

III.1.Basic of Concurrency

In a single-processor, multitasking system, processes/threads are **interleaved** in time to yield the appearance of simultaneous execution. Even though actual parallel processing is not achieved and even though there is a certain amount of overhead involved in switching back and forth between processes/threads, interleaved execution provides major benefits in processing efficiency and in program structuring.

In multiple processor system, it is possible not only to interleave processes/threads but to **overlap** them as well. Both techniques can be viewed as examples of concurrent processing and both present the same problems such as in sharing (global) resources e.g. global variables and in managing the allocation of resources optimally e.g. the request use of a particular I/O channel or device. The following figure try to describe the interleaving the processes. **P** stands for process and **t** is time.

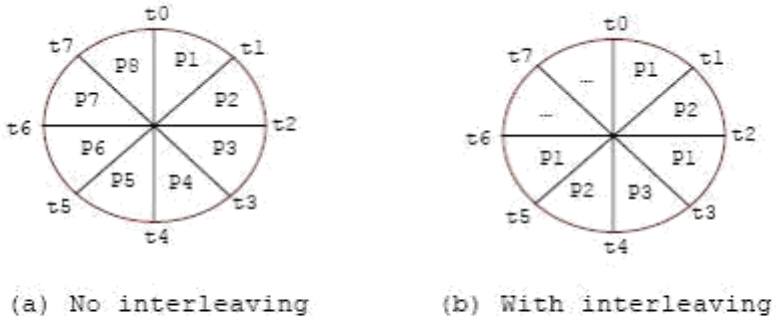


Figure 1: Interleave concept.

III.2.Process Interaction (Interprocess)

In network and distributed computing more process interaction happens. Now let consider these concurrency condition when the processes interact each other instead of a standalone process. Theoretically, from the operating system point of view, these interactions can be classified on the basis of the degree to which processes are aware of each other's existence. These have been summarized in Table 1. Keep in mind that in implementation, several processes may exhibit aspects of both **competition** and **cooperation**.

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other e.g multitasking of multiple independent processes.	Competition for resources e.g. two independent applications may both want access to the same disk or file. Operating system must regulate these accesses.	1. Results of one process independent of the action of others. 2. Timing of process may be affected.	1. Mutual exclusion. 2. Deadlock (for renewable resource) 3. Starvation.
Processes indirectly aware of each other e.g. using shared object such as I/O buffer.	Cooperation by sharing the common object.	1. Result of one process may depend on information obtained from others. 2. Timing of process may be affected.	1. Mutual exclusion. 2. Deadlock for renewable resource. 3. Starvation. 4. Data coherence.
Processes directly aware of each other e.g having communication available to them and are designed to work jointly.	Cooperation by communication.	1. Results of one process may depend on information obtained from others. 2. Timing of process may be affected.	1. Deadlock for consumable resource. 2. Starvation.

Some definitions

1. **Race condition:** In a multithreaded application, a condition that occurs when multiple threads access a data item without coordination, possibly causing inconsistent results, depending on which thread reaches the data item first.

2. **Deadlock:** In multithreaded applications, a threading problem that occurs when each member of a set of threads is waiting for another member of the set. At the end no thread get the resource and all keep waiting.

3. **Concurrency:** The ability of more than one transaction or process to access the same data at the

same time. For the data changes in the database table's cell as an example, this issue must be handled carefully.

4. **Asynchronous call:** A call to a function that is executed separately so that the caller can continue processing instructions without waiting for the function to return.

5. **Synchronous call**: A function call that does not allow further instructions in the calling process to be executed until the function returns. There are two types of file I/O synchronization: synchronous file I/O and asynchronous file I/O. Asynchronous file I/O is also referred to as overlapped I/O.

6. **Synchronous file I/O:** A thread starts an I/O operation and immediately enters a wait state until the I/O request has completed.

7. **Asynchronous file I/O:** A thread performing asynchronous file I/O sends an I/O request to the kernel. If the request is accepted by the kernel, the thread continues processing another job until the kernel signals to the thread that the I/O operation is complete. It then interrupts its current job and processes the data from the I/O operation as necessary.

III.2.1. Competition among processes for resources

Concurrent processes come into conflict with each other when they are competing for the use of the same resource. Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of the other processes and each is to be unaffected by the execution of the other processes. It follows from this that each process should leave the state of any resource that it uses unaffected.

Examples of resource include I/O devices, memory, processor time and the clock. The execution of one process may affect the behavior of competing processes. If two processes both wish access to a single resource then one process will be allocated that resource by the operating system and the other one will have to wait. In an extreme case the blocked process may never get access to the resource and hence will never successfully terminate. In the case of competing processes, three control problems must be solved.

1. The need for **mutual exclusion**. Suppose two or more processes require access to a single non-sharable resource, such as a printer. During the course of execution, each process will be sending

commands to the I/O device, receiving status information, sending data and / or receiving it. We will refer to such a resource as a **critical resource** and the portion of the program that uses it as a **critical section** of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the operating system to understand and enforce this restriction because the detailed requirement may not be obvious. Well, in the case of printer, for example, we wish any individual process to have control of the printer while it prints an entire file else lines from competing processes will be interleaved.

2. Another control problem is a **deadlock** (permanent blocking of a set of processes that either compete for system resources or communicate with each other). Consider two processes P1 and P2 and two **critical resources**, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: R1 is assigned by the operating system to P2, and R2 is assigned to P1. Each process is waiting for one of the two resources. Well, neither will release the resource that it already owns until it has acquired the other resource and performed its critical section. Both processes are deadlocked.

3. Final control problem is **starvation**. Suppose that three processes, P1, P2 and P3, each requires periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that P3 is granted access and that before it completes its critical section, P1 again requires access. If P1 is granted access after P3 has finished, and if P1 and P3 repeatedly grant access to each other, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

Control of competition inevitably involves the operating system because it is the operating system that allocates resources.

III.2.2. Cooperation among processes by sharing

In this case processes that interact with other processes without being explicitly aware of them. For

example, multiple processes may have access to shared variables or to shared files or databases. Processes may use and update the shared data without reference to other processes but know that other processes may have access to the same data. Thus, the processes must **cooperate** to ensure the integrity of the shared data. Because data are held on resources (device, memory) the control problems of mutual exclusion, deadlock and starvation are again present but here the data items may be accesses in two different modes, reading and writing and only writing operation must be mutually exclusive here.

Consider two processes P1 and P2 are sharing data/value A. At time t0, P1 are updating data A to B, and then at t1, P2 are updating data A to C. When P1 reread its previously updated data, well, the data is not accurate anymore (C instead of B). This is also called a **race condition** and there is no data integrity for each process.

III.2.3. Cooperation among processes by communication

Typically, communication can be characterized as consisting some sort of messages. Primitives for sending and receiving messages may be provided as part of the programming language or by the system's kernel of the operating system. Because nothing is shared between processes in the act of passing messages, mutual exclusion is not a control requirement for this sort of cooperation. However the problems of deadlock and starvation are present.

As an example of deadlock, two processes may be blocked, each waiting for a communication from the other. For a starvation, consider three processes P1, P2 and P3, which exhibit the following behavior: P1 is repeatedly attempting to communicate with either P2 or P3, and P2 and P3 are both attempting to communicate with either P1. A sequence could arise in which P1 and P2 exchange information repeatedly, while P3 is blocked waiting for a communication from P1. There is no deadlock because P1 remains active, but P3 is starved.

III.3. Kernel synchronization

You should also understand the architectural basis for these synchronization primitives on multiprocessor systems. Their implementation requires some form of special atomic instructions as a **toehold** for more general synchronization. Examples include test-and-set, compare-and-swap, fetch-

and-add, and load-locked-store-conditional.

Multiprocessor kernels may use these atomic instructions directly for fast and simple primitives to synchronize among processors within the kernel (e.g., spinlocks). You should understand how spinlocks are implemented, performance implications of synchronization with spinlocks, the importance of interrupts and interrupt priority levels on kernel synchronization, and the interaction of spinlocks and interrupts.

Spinlocks can be a basis for implementing the higher-level primitives for synchronizing logical processes or threads (mutexes, condition variables, semaphores). The latter are **blocking** primitives that may transition threads to a blocked (suspended or sleeping) state to prevent the scheduler from running them until some event occurs.

Blocking synchronization primitives must interact directly with the scheduling systems, e.g., using primitives such as the classical Unix kernel primitives **sleep** and **wakeup.** You should understand the performance implications of spinning vs. blocking synchronization.

III.4. Requirement for Mutual Exclusion

The successful implementation of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion. This is fundamental for any concurrent processing scheme. Generally, any facility or capability that is to provide support for mutual exclusion should meet the following requirement:

1. Mutual exclusion must be entered: Only one process at a time is allowed into its critical section among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non-critical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely; no deadlock or starvation can be allowed.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processes.
6. A process remains inside its critical section for a finite time only.

III.5. Synchronization in Windows Operating Systems

To avoid **race conditions**, **deadlocks** and other related conditions as discussed before, it is necessary to synchronize access by multiple threads to shared resources. Synchronization is also necessary to ensure that interdependent code is executed in the proper sequence.
In Windows, to synchronize access to a resource, you can use one of the **synchronization object**s in one of the **wait functions**. Each synchronization object instance can be in either a **signaled** or **non-signaled state**. A thread can be **suspended** on an object in a non-signaled state; the thread is **released** when the object enters the signaled state.

The mechanism is a thread issues a **wait** request to the NT executive by using the handle of the synchronization object. When an object enters the signaled state, the NT executive releases all thread objects that are waiting on the synchronization object. The **wait functions** allow a thread to block its own execution until a specified non-signaled object is set to the signaled state. The functions described in this section provide mechanisms that threads can use to synchronize access to a resource.

III.5.1. Synchronization Objects

A synchronization object is an object whose **handle** can be specified in one of the **wait functions to coordinate** the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible.

In Windows Win32 programming, the following object types are provided exclusively for synchronization.

In some circumstances, you can also use a **file**, **named pipe**, or **communications device** as a synchronization object; however, their use for this purpose is discouraged. Instead, use asynchronous I/O and wait on the **event object** set in the overlapped structure. It is safer to use the event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

III.5.2. Wait Functions

The wait functions allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters the wait state. It uses no processor time while waiting for the criteria to be met.

There are four types of wait functions:

1. Single-object.
2. Multiple-object.
3. Alertable.
4. Registered.

Single-object Wait Functions

The SignalObjectAndWait(), WaitForSingleObject(), and WaitForSingleObjectEx() functions require a handle to one synchronization object. These functions return when one of the following occurs:

1. The specified object is in the signaled state.
2. The time-out interval elapses. The time-out interval can be set to **INFINITE** to specify that the wait will not time out.

The SignalObjectAndWait() function enables the calling thread to atomically set the state of an object to signaled and wait for the state of another object to be set to signaled.

Multiple-object Wait Functions

The WaitForMultipleObjects(), WaitForMultipleObjectsEx(), MsgWaitForMultipleObjects(), and MsgWaitForMultipleObjectsEx() functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following occurs:

1. The state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled. You control whether one or all of the states will be used in the function call.

2. The time-out interval elapses. The time-out interval can be set to **INFINITE** to specify that the wait will not time out.

The MsgWaitForMultipleObjects() and MsgWaitForMultipleObjectsEx() function allow you to specify input event objects in the object handle array. This is done when you specify the type of input to wait for in the thread's input queue. For example, a thread could use MsgWaitForMultipleObjects() to block its execution until the state of a specified object has been set to signaled and there is mouse input available in the thread's input queue.
The thread can use the GetMessage() or PeekMessage() function to retrieve the input. When waiting for the states of all objects to be set to signaled, these multiple-object functions do not modify the states of the specified objects until the states of all objects have been set signaled. For example, the state of a mutex object can be signaled, but the calling thread does not get ownership until the states of the other objects specified in the array have also been set to signaled. In the meantime, some other thread may get ownership of the mutex object, thereby setting its state to non-signaled.

Alertable Wait Functions

The MsgWaitForMultipleObjectsEx(), SignalObjectAndWait(), WaitForMultipleObjectsEx(), and WaitForSingleObjectEx() functions differ from the other wait functions in that they can optionally perform an alertable wait operation. In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread.

Registered Wait Functions

The RegisterWaitForSingleObject() function differs from the other wait functions in that the wait operation is performed by a thread from the **thread pool**. When the specified conditions are met, the callback function is executed by a worker thread from the thread pool. By default, a registered wait operation is a multiple-wait operation. The system resets the timer every time the event is signaled (or the time-out interval elapses) until you call the UnregisterWaitEx() function to cancel the operation. To specify that a wait operation should be executed only once, set the dwFlags parameter of RegisterWaitForSingleObject() to WT_EXECUTEONLYONCE.

Wait Functions and Synchronization Objects

The wait functions can modify the states of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. Wait functions can modify the states of synchronization objects as follows:

1. The count of a semaphore object decreases by one, and the state of the semaphore is set to non-signaled if its count is zero.
2. The states of mutex, auto-reset event, and change-notification objects are set to non-signaled.
3. The state of a synchronization timer is set to non-signaled.
4. The states of manual-reset event, manual-reset timer, process, thread, and console input objects are not affected by a wait function.

Wait Functions and Creating Windows

You have to be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. Two examples of code that indirectly creates windows are DDE and the CoInitialize() function.

Therefore, if you have a thread that creates windows, use MsgWaitForMultipleObjects() or MsgWaitForMultipleObjectsEx(), rather than the other wait functions.

III.6 Synchronization in UNIX Operating System

Many processes can execute concurrently on the UNIX Operating system (Multiprogramming or multitasking) with no limit to their number logically. Some system calls allow processes execution, and control reaction to various events.

In UNIX System, the only active entities are the processes. Each process runs a single program and initially has a single thread of control. Process management of the UNIX operating system is typically responsible for tasks: to create, suspend and terminated process, to switch their states, to schedule the CPU to execute multiple processes concurrently in time sharing system, and to manage the communication between processes

III.6.1 Thread Synchronization in UNIX

A thread is a single sequence stream within in a process. When multiple threads share the same memory, we need to make sure that each thread sees a consistent view of its data. If each thread uses variables that other threads don't read or modify, no consistency problems will exist. Similarly, if a variable is read-only, there is no consistency problem with more than one thread reading its value at the same time. However, when one thread can modify a variable that other threads can read or modify, we need to synchronize the threads to ensure that they don't use an invalid value when accessing the variable's memory contents.

Thread synchronization requires that a running thread gain a **lock** on an object before it can access it. The thread will wait in line for another thread that is using the method/data member to be done with it. This is very important to prevent the corruption of program data if multiple threads will be accessing the same data. If two threads try to change a variable or execute the same method at the same, this can cause serious and difficult to find problems. Thread synchronization helps prevent this.

III.6.2 Process synchronization in UNIX

A process is an instance of a program in execution. Since processes in concurrent systems frequently need to communicate with other processes therefore, there is a need for a well-structured communication, without using interrupts, among processes. Processes use two kinds of synchronization to control their activities;

a) **Control synchronization:** it is needed if a process waits to perform some action only after some other processes have executed some action,

b) **Data access synchronization:** It is used to access shared data in a mutually exclusive manner. The basic technique used to implement this synchronization is to block a process until an appropriate condition is fulfilled. The synchronization of concurrent processes is great importance in multiprocessing operating systems.

III.6.3 Means of synchronization in UNIX operating system

- Mutexes
- Semaphores
- Monitors
- Condition variables
- Barrier
- Spin lock

1. Mutexes

Threads can synchronize using locks called **mutexes**. It is possible to protect data and ensure access by only one thread at a time by using the pthreads mutual-exclusion interfaces. A mutex is basically a lock that we set (lock) before accessing a shared resource and release (unlock) when we're done. While it is set, any other thread that tries to set it will block until we release it. If more than one thread is blocked when we unlock the mutex, then all threads blocked on the lock will be made runnable, and the first one to run will be able to set the lock. The others will see that the mutex is still locked and go back to waiting for it to become available again. In this way, only one thread will proceed at a time.

2. Spin Locks

A spin lock is like a mutex, except that instead of blocking a process by sleeping, the process is blocked by busy-waiting (spinning) until the lock can be acquired. A spin lock could be used in situations where locks are held for short periods of times and threads don't want to incur the cost of being descheduled.

Spin locks are often used as low-level primitives to implement other types of locks. Depending on the system architecture, they can be implemented efficiently using test-and-set instructions. Although

efficient, they can lead to wasting CPU resources: while a thread is spinning and waiting for a lock to become available, the CPU can't do anything else. This is why spin locks should be held only for short periods of time.

3. Semaphores

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V.

Initialization operation called **Semaphoiinitislize**. Binary Semaphores can assume only the value 0 or the value 1, counting semaphores, also called general semaphores, can assume only nonnegative values. P() or down() decrements the semaphore by one. If the semaphore is zero, the process calling P() is blocked until the semaphore is positive again. The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S).

V() or up() increments the semaphore by one. The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S).

Note: Semaphore *uses* the wait and signal operations. A process that has to wait should be put to sleep, and should wake up only when a corresponding signal occurs, as that is the only time the process has any chance to proceed.

4. Condition Variables

Condition variables are another synchronization mechanism available to threads. These synchronization objects provide a place for threads to rendez-vous. When used with mutexes, condition variables allow threads to wait in a race-free way for arbitrary conditions to occur.

The condition itself is protected by a mutex. A thread must first lock the mutex to change the condition state. Other threads will not notice the change until they acquire the mutex, because the mutex must be locked to be able to evaluate the condition.

5. Monitors

Monitor is higher level synchronization. A monitor objects holds several condition variables (waiting rooms)

Operations:

- Custom operations (fully synchronized)
- Wait(): release the monitor and wait
- Notify(): wake up one of the waiting processes
- Notify All(): wake up all the waiting processes

6. Barriers

Barriers are a synchronization mechanism that can be used to coordinate multiple threads working in parallel. A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there. They allow an arbitrary number of threads to wait until all of the threads have completed processing, but the threads don’t have to exit. They can continue working after all threads have reached the barrier.

CHAPTER IV: PARALLEL PROGRAMMING

Principles of parallel programming

With the rise of multi-core architecture, parallel programming is an increasingly important topic for software engineers and computer system designers. Written by well-known researchers Larry Snyder and Calvin Lin, this highly anticipated first edition emphasizes the principles underlying parallel computation, explains the various phenomena, and clarifies why these phenomena represent opportunities or barriers to successful parallel programming

Variable definitions

Mutable: values may be assigned to the variables and changed during program execution (as in sequential languages).

Definitional: variable may be assigned a value only once

A **parallel programming model** is a concept that enables the expression of parallel programs which can be compiled and executed. The value of a programming model is usually judged on its generality, how well a range of different problems can be expressed and how well they execute on a range of different architectures. The implementation of a programming model can take several forms such as libraries invoked from traditional sequential languages, language extensions, or complete new execution models.

Consensus on a particular programming model is important as it enables software expressed within it to be transportable between different architectures. The von Neumann model has facilitated this with sequential architectures as it provides an efficient bridge between hardware and software, meaning that high-level languages can be efficiently compiled to it and it can be efficiently implemented in hardware.

Issues in parallel programming not found in sequential programming

- Task decomposition, allocation and sequencing

Breaking down the problem into smaller tasks (processes) than can be run in parallel

Allocating the parallel tasks to different processors, Sequencing the tasks in the proper order, efficiently use the processors

- Communication of interim results between processors

The goal is to reduce the cost of communication between processors. Task decomposition and allocation affect communication costs

- Synchronization of processes

Some processes must wait at predetermined points for results from other processes.

- Different machine architectures

CHAPTER V: PARALLEL ALGORITHMS AND COMPLEXITY.

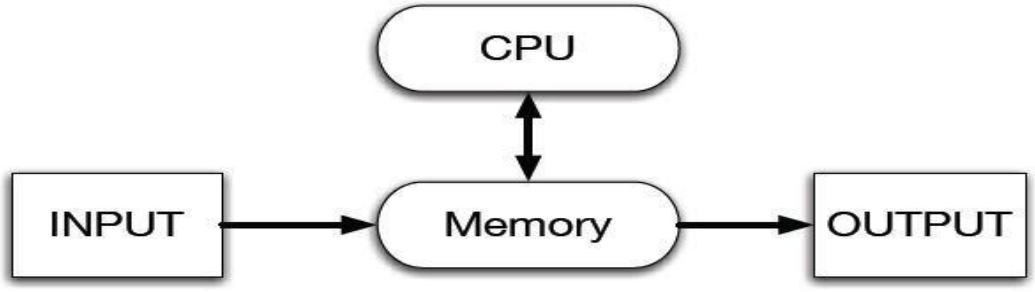
V.1. What is an algorithm?

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

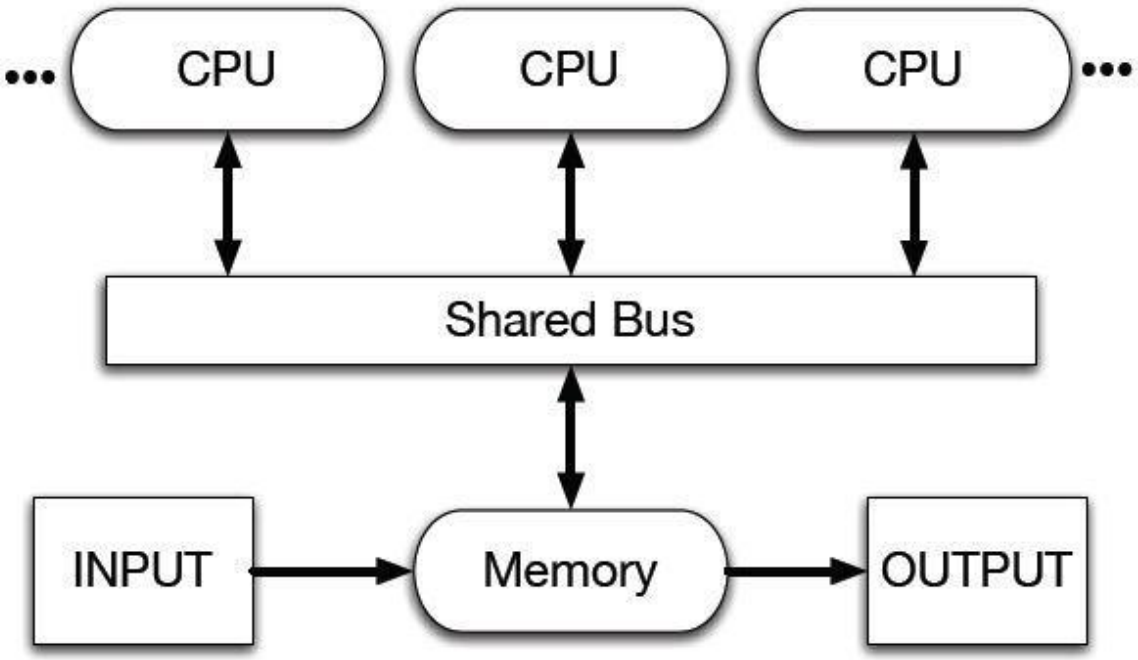
We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.

An algorithm is said to be *correct* if, for every input instance, it halts with the correct output. We say that a correct algorithm *solves* the given computational problem. An incorrect algorithm might not halt at all on some input instances, or it might halt with other than the desired answer. Contrary to what one might expect, incorrect algorithms can sometimes be useful, if their error rate can be controlled. Concerned only with correct algorithms. An algorithm can be specified in English, as a computer program, or even as hardware

The only requirement is that the specification must provide a precise description of the computational procedure to be followed. In looking at the development of parallel algorithms, the standard Von Neumann architecture is modified, from this Logical View.



To this Logical View



Unlimited CPU unlimited memory bus details left unspecified

V.2 The parallel algorithms

The parallel algorithms are composed from existing functionality in the Concurrency Runtime.

V.2.1 Parallel_for algorithm

The concurrency::parallel_for algorithm uses a concurrency::structured_task_group object to perform the parallel loop iterations. The parallel_for algorithm partitions work in an optimal way given the available number of computing resources.

The concurrency::parallel_for algorithm repeatedly performs the same task in parallel. Each of these tasks is parameterized by an iteration value. This algorithm is useful when you have a loop body that does not share resources among iterations of that loop.

The parallel_for algorithm partitions tasks in an optimum way for parallel execution. It uses a work-

stealing algorithm and range stealing to balance these partitions when workloads are unbalanced. When one loop iteration blocks cooperatively, the runtime redistributes the range of iterations that is assigned to the current thread to other threads or processors. Similarly, when a thread completes a range of iterations, the runtime redistributes work from other threads to that thread. The parallel_for algorithm also supports nested parallelism.

- The parallel_for algorithm does not execute the tasks in a pre-determined order.
- The parallel_for algorithm does not support arbitrary termination conditions.
- The parallel_for algorithm stops when the current value of the iteration variable is one less than _Last.

V.2.2 The parallel_invoke Algorithm

The **concurrency::parallel_invoke** algorithm executes a set of tasks in parallel. It does not return until each task finishes. This algorithm is useful when you have several independent tasks that you want to execute at the same time.The **parallel_invoke** algorithm takes as its parameters a series of work functions (lambda functions, function objects, or function pointers). The **parallel_invoke** algorithm is overloaded to take between two and ten parameters. Every function that you pass to **parallel_invoke** must take zero parameters.

Like other parallel algorithms, **parallel_invoke** does not execute the tasks in a specific order. The topic **Task Parallelism (Concurrency Runtime)** explains how the **parallel_invoke** algorithm relates to tasks and task groups.

V.2.3 Parallel Random-Access Machine (PRAM Model)

A **parallel random-access machine** is a shared-memory abstract machine. There are n ordinary (serial) processors that have a shared, global memory. All processors can read from or write to the global memory in parallel (at the same time). The processors can also perform various arithmetic and logical operations in parallel. Running time can be measured as the number of parallel memory accesses an algorithm performs. As its name indicates, the PRAM was intended as the parallel-computing analogy to the random-access machine (RAM). In the same way that the RAM is used by sequential-algorithm designers to model algorithmic performance (such as time complexity), the PRAM is used by parallel-

algorithm designers to model parallel algorithmic performance (such as time complexity, where the number of processors assumed is typically also stated). Similar to the way in which the RAM model neglects practical issues, such as access time to cache memory versus main memory, the PRAM model neglects such issues as synchronization and communication, but provides any (problem-size-dependent) number of processors. Algorithm cost, for instance, is estimated using two parameters O(time) and O(time × processor_number). Real parallel computers cannot perform parallel accesses to global memory in unit time. The time for a memory access grows with the number of processors in the parallel computer. However, real parallel machines typically have a communication network that can support the abstraction of a global memory. Accessing data through the network is a relatively slow operation in comparison with arithmetic and other operations.

Thus, counting the number of parallel memory accesses executed by two parallel algorithms does yield a fairly accurate estimate of their relative performances.

V.2.4 Read/write conflicts

Read/write conflicts in accessing the same shared memory location simultaneously are resolved by one of the following strategies:

1. Exclusive read exclusive write (EREW)—every memory cell can be read or written to by only one processor at a time
2. Concurrent read exclusive write (CREW)—multiple processors can read a memory cell but only one can write at a time
3. Exclusive read concurrent write (ERCW)—never considered
4. Concurrent read concurrent write (CRCW)—multiple processors can read and write. A CRCW PRAM is sometimes called a **concurrent random-access machine.**

It is up to algorithm to enforce the chosen model including how to resolve two or more writes to the same location. Another kind of *array reduction* operation like SUM, Logical AND or MAX. Several simplifying assumptions are made while considering the development of algorithms for PRAM. They are:

1. There is no limit on the number of processors in the machine.
2. Any memory location is uniformly accessible from any processor.
3. There is no limit on the amount of shared memory in the system.
4. Resource contention is absent.
5. The programs written on these machines are, in general, of type **MIMD.** Certain special cases such as **SIMD** may also be handled in such a framework.

These kinds of algorithms are useful for understanding the exploitation of concurrency, dividing the original problem into similar sub-problems and solving them in parallel.

Concurrent read access and exclusive write access to data in shared memory architecture

Concurrent read access and exclusive write access are provided in a shared memory architecture to permit one or more devices in the shared memory architecture to maintain read access to a block of memory such as a cache line while one device has exclusive permission to modify that block of memory. By doing so, a device that has permission to modify may make updates to its copy of the block of memory without invalidating other copies of the block of memory, and potentially enabling other devices to continue to read data from their respective copies of the block of memory without having to retrieve the updated copy of the block of memory.

V.2.5 Pointer Jumping

The technique of pointer jumping (or *pointer doubling*) allows fast parallel processing of linked data structures such as lists and trees. We usually draw trees with edges directed from children to parents.

V.2.6 Brent's Theorem and work efficiency

The following theorem, due to Brent, relates the work and time complexities of a parallel algorithm described in the WT formalism to its running time on a p-processor PRAM.

V.3. Parallel complexity theory

A common sense on feasibility of some computing resource is that the growth rate for the resource is bounded by a polynomial in input size. In the world of sequential problem solving, the most valuable resource is time. Hence, a problem is feasible, or tractable, if it is solvable in polynomial time and intractable if the best known algorithm has higher than polynomial time complexity and all other intractable problems are known to be reducible to this problem. (Recent advances in complexity theory consider approximatively tractable problems which are intractable if we seek exact solutions, but are tractable if we can accept approximate solutions).
Obviously, a sequentially intractable problem remains intractable in parallel if we use at most polynomial number of processors. And polynomial number of processors is the maximum we can hope for in practice. Hence, parallel complexity theory deals only with sequential polynomial problems. Most of the machinery of the parallel complexity theory is derived from the sequential one, which uses specific formal framework to make its results and conclusions independent on particular implementation details of algorithms and robust with respect to various models and architectures of computing devices. So before explain the basic ideas of the parallel complexity, we review the basic notions of the classical complexity theory, so that we can understand how the parallel complexity theory relates to the classical one.

Basics notions of sequential complexity theory

Complexity class: a collection of problems of some characteristic worst cases difficulty. Some problems in a class may be easier than the others, but all of them can be solved within the same resource bounds associated with the class.

Abstract problem: a binary relation between a set of problem instances and a set of the problem solutions. The size of an instance is the number of symbols (digits, letters, delimiters, etc) in the full problem specification.

Example: SPP = Shortest Path Problem: Given graph G and two vertices u,v, find in G a shortest path between u and v. The set of instances is the set of all triples (G,u,v) and a given instance may have zero, one, or more solutions - paths from u to v.

Decision problem: a problem with solutions Yes/No. Optimization problems can be formulated as decision problems, typically by imposing bounds on the values to be optimized.
Example: PATH = the decision problem for SPP: Given G, u,v and an integer k, does a path exist in G between u and v whose length is at most k?

Observation: If a decision problem is hard, so is the related optimization problem (since easy optimization problem makes the decision problem easy). Therefore, the complexity theory deals only with decision problems.

Compact encoding: Any abstract notions, such as numbers, graphs, functions, polygons, programs, can be encoded using strings over some finite alphabet Z. An encoding is **compact** if the length of the encoded instance is of the same order as the size of the abstract instance. That is, they must be equivalent within a polynomial factor.

Concrete problem: an abstract problem **encoded** for a computing device using some **compact** encoding, typically a **binary** one (such as ASCII).
Input size: the length of a binary encoding of a problem instance, denoted by $|x|$ for input x.

Sequential time: A sequential algorithm solves a concrete problem in time $T(n)$ if it produces a solution in time $O(T(|x|))$ for any input x.
Polynomial class *P*: the set of all problems with $T(n)=O(n^{O(1)})$. A polynomial problem remains polynomial regardless of which ``compact" encoding is chosen, since any such encoding can be converted into a binary one in polynomial time. That is why the complexity theory deals only with binary encoding.

Input accepted/rejected by an algorithm: Algorithm A **accepts** (**rejects**) binary string x if A outputs 1 (0) for input x.
Acceptability: L is **accepted** in time $T(n)$ if any x *in* L is accepted in time $T(n)$. To accept a language, the algorithm needs only to worry about strings in the language. If x is not accepted, then it is either rejected or A may loop forever for input x.

PART TWO: DISTRIBUTED SYSTEMS

CHAPTER I: INTRODUCTION

I.1 What is a distributed system?

A distributed system is collection of heterogeneous nodes connected by one or more interconnection networks which provides access to system-wide shared resources and services. It is a collection of independent computers that appears to its users as a single coherent system.

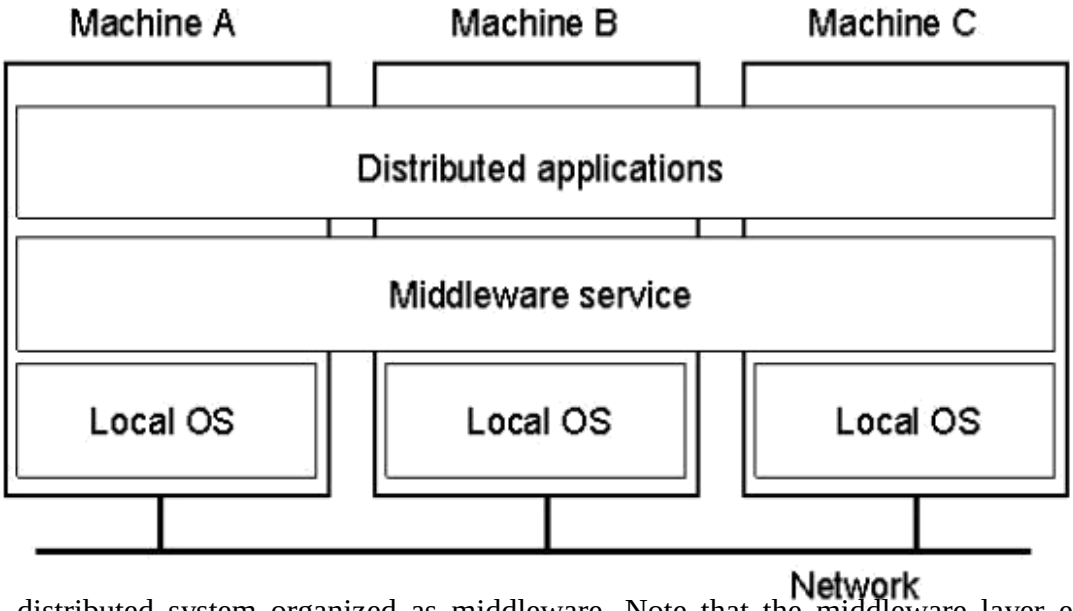
A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. This software enables computers to coordinate their activities and to share the resources of the system hardware, software, and data. Users of a distributed system should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations. This is in contrast to a network, where the user is aware that there are several machines whose locations, storage replications, load balancing, and functionality are not transparent.

Benefits of distributed systems include bridging geographic distances, improving performance and availability, maintaining autonomy, reducing cost, and allowing for interaction.

I.2 Characteristics of a distributed system.

Multiple Computers: More than one physical computer, each consisting of CPUs, local memory, and possibly stable storage, and I/O paths to connect it with the environment.
Interconnections: Mechanisms for communicating with other nodes via a network.
Shared State: If a subset of nodes cooperates to provide a service, a shared state is maintained by these nodes. The shared state is distributed or replicated among the participants.

An Abstract View Distributed Systems



A distributed system organized as middleware. Note that the middleware layer extends over multiple machines.

I.3 Distributed vs. Centralized Systems

- Why distribute?
- Resource sharing
- Device sharing
- Flexibility to spread load
- Incremental growth
- Cost/performance
- Reliability/Availability
- Inherent distribution
- Security?

Why NOT distribute(challenge)?

Software,Network,Security,System management

Numerous sources of complexity including:

Transparent/uniform access to data or resources
Independent failures of processors (or processes)

Dynamic membership of a system

Unreliable/unsecured communication

I.4 Design Goals & Issues

Connecting users and resources is the primary goal

Transparency: hide the fact that processes and resources are physically distributed

Openness: offer services according to rules and interfaces that describe the syntax and semantics of those services

Interoperability and portability, Separating policy from mechanism, Overhead in message communication, Distribution or replication of data (or meta-data), Lack of clean common interfaces.

The object-oriented model for a distributed system is based on the model supported by object-oriented programming languages. Distributed object systems generally provide remote method invocation (RMI) in an object-oriented programming language together with operating systems support for object sharing and persistence. Remote procedure calls, which are used in client-server communication, are replaced by remote method invocation in distributed object systems.

The state of an object consists of the values of its instance variables. In the object-oriented paradigm, the state of a program is partitioned into separate parts, each of which is associated with an object. Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension. The Object Management Group's Common Object Request Broker (CORBA) is a widely used standard for distributed object systems. Other object management systems include the Open Software Foundation's Distributed Computing Environment (DCE) and Microsoft's Distributed

Common Object Manager (DCOM).

I.5 Distributed systems (control systems)

Collections of modules, each with its own specific function, interconnected to carry out integrated data acquisition and control. Industrial control systems have evolved from totally analog systems through centralized digital computer-based systems to multilevel, distributed systems. Originally, industrial control systems were entirely analog, with each individual process variable controlled by a single feedback controller. Although analog control systems were simple and reliable, they lacked integrated information displays for the process operator.

In supervisory control, the analog portion of the system is implemented in a traditional manner (including analog display in the central operating room), but a digital computer is added which periodically scans, digitizes, and inputs process variables to the computer. The computer is used to filter the data, compute trends, generate specialized displays, plot curves, and compute unmeasurable quantities of interest such as efficiency or quality measures. Once such data are available, optimal operation of the process may be computed and implemented by using the computer to output set-point values to the analog controllers. This mode of control is called supervisory control because the computer itself is not directly involved in the dynamic feedback.

Direct digital control replaces the analog control with a periodically executed equivalent digital control algorithm carried out in the central digital computer. A direct digital control system periodically scans and digitizes process variables and calculates the change required in the manipulated variable to reduce the difference between the set point and the process variable to zero.

The advantages of direct digital control are the ease with which complex dynamic control functions can be carried out and the elimination of the cost of the analog controllers themselves. To maintain the attractive display associated with analog control systems, the display portion of the analog controller is usually provided. Thus operation of the process is identical to operation of digital supervisory control systems with analog controllers, except that “tuning” of the controllers (setting of gains) can be done through operator consoles.

The cost reduction which resulted from the introduction of direct digital control was offset by a number of disadvantages. The most notable of these were the decrease in reliability and the total loss of graceful degradation. Failure of a sensor or transmitter had the same effect as before, but failure of the computer itself threw the entire control system into manual operation. Hence it was necessary to provide analog controllers to back up certain critical loops which had to function even when the computer was down.

Increasing demand for ever-higher levels of supervisory control highlighted two disadvantages of centralized digital computer control of processes. First, process signals were still being transmitted from the process sensor to the central control room in analog form, meaning that separate wires had to be installed for every signal going to or from the computer. Second, the digital computer system itself evolved into a very complex unit because of the number of devices attached to the computer and because of the variety of different programs needed to carry out the myriad control and management functions. The latter resulted in the need for an elaborate real-time operating system for the computer which could handle resources, achieve desired response time for each task, and be responsible for error detection and error recovery in a highly dynamic real-time environment. Design coding, installation, and checkout of centralized digital control systems was so costly and time-consuming that application of centralized digital control was limited.

Low-cost electronic hardware utilizing large-scale integrated circuits provided the technology to solve both of these problems while retaining the advantages of centralized direct digital and supervisory control. The solution, distributed control, involved distributing control functions among hardware modules to eliminate the critical central computer.

The combination of reliable, responsive distributed control and general-purpose communication networks leads to a system which can be adapted to critical control applications in a very flexible manner, with potential for increased productivity in plants, increased safety, and decreased energy consumption. Technology for higher-speed computation, data communication, and object-oriented software organization allows the integration of distributed control systems into plant-wide and enterprise-wise systems.

CHAPTER II: CLIENT/SERVER

Server is a computer that supplies services or data to other machines on a local area network (LAN) or a wide area network (WAN) such as the Internet. Some servers run administrative software that controls access to all or part of the network and its resources (such as disk drives or printers). Others provide files, applications, or World Wide Web pages. Computers that request services or data from a server are known as **clients.**

II.1 Client/Server Architecture

An arrangement used on local area networks that makes use of "distributed intelligence" to treat both the server and the individual workstations as intelligent, programmable devices, thus exploiting the full computing power of each. This is done by splitting the processing of an application between two distinct components: a "front-end" client and a "back-end" server. The client component, itself a complete, stand-alone personal computer (versus the "dumb" terminal found in older architectures such as the time-sharing used on a mainframe) offers the user its full range of power and features for running applications.

The server component, which can be another personal computer, minicomputer, or a mainframe, enhances the client component by providing the traditional strengths offered by minicomputers and mainframes in a time-sharing environment: data management, information sharing between clients, and sophisticated network administration and security features. The advantage of the client/server architecture over older architectures is that the client and server machines work together to accomplish the processing of the application being used. Not only does this increase the processing power available, but it also uses that power more efficiently. The client portion of the application is typically optimized for user interaction, whereas the server portion provides the centralized, multi-user functionality.

II.2 Distributed Processing

Distributed Processing is a form of information processing in which work is performed by separate computers that are linked through a communications network. Distributed processing is

usually categorized as either **plain distributed processing** or **true distributed processing**.

- Plain distributed processing shares the workload among computers that can communicate with one another.
- True distributed processing has separate computers perform different tasks in such a way that their combined work can contribute to a larger goal, such as the transfer of funds from one bank to another. This type of processing requires a highly structured environment that allows hardware and software to communicate, share resources, and exchange information freely. At the highest (and most visible) levels, such distributed processing can also require data-transfer mechanisms that are relatively invisible to users but that enable different programs to use and share one another's data.

II.2.1 Message Passing

Message passing is the basis of most interprocess communication in distributed systems. It is at the lowest level of abstraction and requires the application programmer to be able to identify the destination process, the message, the source process and the data types expected from these processes.

1. **Syntax**

Communication in the message passing paradigm, in its simplest form, is performed using the **send()** and **receive()** primitives. The syntax is generally of the form: **send**(*receiver, message*)
receive(*sender, message*)

The **send()** primitive requires the name of the destination process and the message data as parameters. The addition of the name of the sender as a parameter for the **send()** primitive would enable the receiver to acknowledge the message. The **receive()** primitive requires the name of the anticipated sender and should provide a storage buffer for the message.

2. **Semantics**

Decisions have to be made, at the operating system level, regarding the semantics of the **send()** and **receive()** primitives. The most fundamental of these are the choices between blocking and non-blocking primitives and reliable and unreliable primitives.

Blocking/non-blocking: A blocking **send()** blocks the process and does not execute the following instruction until the message has been sent and the message buffer has been cleared. In

the same way a blocking receive blocks at the **receive()** until the message arrives. A non blocking send returns control to the caller immediately. The message transmission is then executed concurrently with the sending process. This has the advantage of not leaving the CPU idle while the send is being completed. However, the disadvantage of this approach is that the sender does not know and will not be informed when the message buffer has been cleared. To overcome this the kernel can either make a copy of the message buffer or send an interrupt to the sender when the message buffer has been cleared. At the implementation level, although non-blocking primitives are flexible they make programming and debugging very difficult, hence, for the sake of easier programming, blocking primitives are often chosen.

Buffered/unbuffered messages: An unbuffered **receive()** means that the sending Process sends the message directly to the receiving process rather than a message buffer. The address, *receiver,* in the **send()** is the address of the process, but in the case of an buffered **send()** the address is that of the buffer. There is a problem, in the unbuffered case, if the **send()** is called before the **receive()** because the address in the send does not refer to any existing process on the server machine. Buffered messages are saved in a buffer until the server process is ready to receive them. They can best be implemented through a mechanism in the operating system which can keep a backlog of **sends**, a port in which messages are queued waiting until requested by the receiver. The buffer capacity can either be bounded, where a predetermined number of messages can be stored, or unbounded. An unbounded buffer could be implemented using dynamic memory allocation, thus its capacity would be fixed only by the size of available memory.

Reliable/unreliable send: Unreliable **send()** sends a message to the receiver and does not expect acknowledgement of receipt, nor does it automatically retransmit the message to ensure receipt. A reliable **send ()** guarantees that, by the time the **send()** is complete, the message has been received. The primitive itself handles acknowledgements and retransmission in response to lost messages. At the implementation level the operating system must wait only for a specified length of time, so that a process does not remain blocked indefinitely waiting for a response from a receiver that has terminated. Lost messages are handled either by the operating system retransmitting the message or informing the sender of the message's loss or by the sender detecting the loss itself.

Direct/indirect communication: Ports allow indirect communication. Messages are sent to the port by the sender and received from the port by the receiver. Direct communication involves the message being sent direct to the process itself, which is named explicitly in the send, rather than to the intermediate port.

Fixed/variable size messages: Fixed size messages have their size restricted by the system. The implementation of variable size messages is more difficult but makes Programming easier; the reverse is true for fixed size messages.

Passing data by reference/value/address mapping: Data, in message passing, is often passed by value, since the processes execute in separate address spaces However, another parameter passing mechanism is available which would be suitable for a message passing system. This is referred to as call-by-copy/restore. The variable is essentially passed by value but the returned value overwrites the original value so that the final result is the same as if it were passed by reference In message passing systems the onus is on the application programmer to control data movement between processes and to control the synchronization of these processes, where they have access to shared data.

II.2.2 Microsoft Remote Procedure Call (RPC)

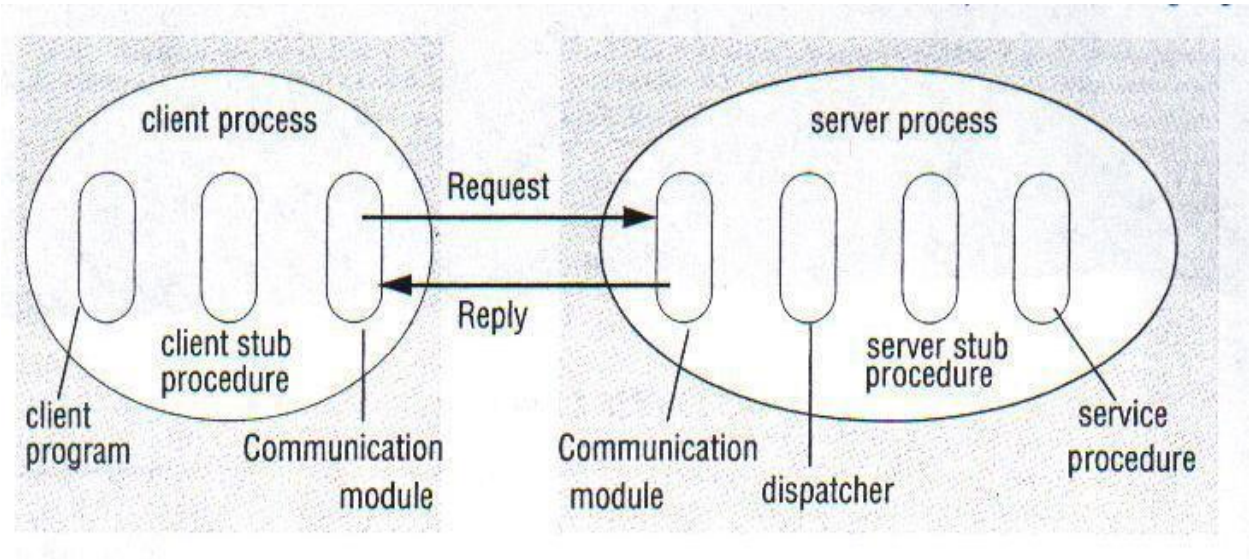
Microsoft Remote Procedure Call (RPC) is a powerful technology for creating distributed client/server programs. RPC is an interprocess communication technique that allows client and server software to communicate. The Microsoft RPC facility is compatible with the Open Group's Distributed Computing Environment (DCE) specification for remote procedure calls and is interoperable with other DCE-based RPC systems, such as those for HP-UX and IBM AIX UNIX–based operating systems.

Computer operating systems and programs have steadily gotten more complex over the years. With each release, there are more features. The growing intricacy of systems makes it more difficult for developers to avoid errors during the development process. Often, developers create a solution for their system or application when a nearly identical solution has already been devised. This duplication of effort consumes time and money and adds complexity to already

complex systems.

RPC is designed to mitigate these issues by providing a common interface between applications. RPC serves as a go–between for client/server communications. RPC is designed to make client/server interaction easier and safer by factoring out common tasks, such as security, synchronization, and data flow handling, into a common library so that developers do not have to dedicate the time and effort into developing their own solutions.

Figure 1. Role of client and server stub procedures in RPC in the context of a procedural language



- RPC only addresses procedure calls.
- RPC is not concerned with objects and object references.
- A client that accesses a server includes one stub procedure for each procedure in the service interface.
- A client stub procedure is similar to a proxy method of RMI.
- A server stub procedure is similar to a skeleton method of RMI.
- Message passing leaves the programmer with the burden of the explicit control of the movement of data. Remote procedure calls (RPC) relieves this burden by
- increasing the level of abstraction and providing semantics similar to a local procedure call.

1 Syntax

The syntax of a remote procedure call is generally of the form:

call procedure_name(*value_arguments; result_arguments*)

The client process blocks at the **call()** until the reply is received. The remote procedure is the server processes which has already begun executing on a remote machine.

It blocks at the **receive()** until it receives a message and parameters from the sender. The server then sends a **reply**() when it has finished its task. The syntax is as follows: **receive** procedure_name(*in value_parameters; out result_parameters*)

reply (*caller, result_parameters*)

2 .Semantics

The semantics (meaning) of RPC are the same as those of a local procedure call — the calling process calls and passes arguments to the procedure and it blocks while the procedure executes. When the procedure completes it can return results to the calling process. In the simplest case, the execution of the **call()** generates a client stub which organizes the arguments into a message and sends the message to the server machine. On the server machine the server is blocked awaiting the message. On receipt of the message the server stub is generated and extracts the parameters from the message and passes the parameters and control to the procedure. The results are returned to the client with the same procedure in reverse .The following issues regarding the properties of remote procedure calls need to be considered in the design of an RPC system if the distributed system is to achieve transparency.

Binding: Binding provides a connection between the name used by the calling process and the location of the remote procedure. Binding can be implemented, at the operating system level, using a static or dynamic linker extension which binds the procedure name with its location on another machine. Another method is to use procedure variables which contain a value which is linked to the procedure location.

Communication transparency: The users should be unaware that the procedure they are calling is remote. The three difficulties when attempting to achieve transparency are: the detection and correction of errors due to communication and site failures, the passing of parameters, and exception handling. Communication and site failures can result in inconsistent data because of partially completed processes. The solution to this problem is often left to the application

programmer. Parameter passing in most systems is restricted to the use of value parameters. Exception handling is a problem also associated with heterogeneity. The exceptions available in different languages vary and have to be limited to the lowest common denominator.

Concurrency: Concurrency mechanisms should not interfere with communication mechanisms. Single threaded clients and servers, when blocked while waiting for the resultsNfrom a RPC, can cause significant delays. These delays can be exacerbated by further remote procedure calls made in the server. Lightweight processes allow the server to execute calls from more than one client concurrently.

Heterogeneity**:** Different machines may have different data representations, the Machines may be running different operating system or the remote procedure may have been written using a different language. Static interface declarations of remote procedures serve to establish agreement between the communicating processes on argument types, exception types (if included), type checking and automatic conversion from one data representation to another, where required. Generally RPC proves a simpler means for an application programmer to construct distributed programs than simple message passing because it abstracts away from the details of communication and transmission. However, the achievement of true transparency is a problem which has not been completely resolved for RPC, still leaving much of the work and responsibility for the application programmer.

Terms and Definitions

The following terms are associated with RPC.

Client: A process, such as a program or task, that requests a service provided by another program. The client process uses the requested service without having to “deal” with many working details about the other program or the service.

Server: A process, such as a program or task, that responds to requests from a client.

Endpoint: The name, port, or group of ports on a host system that is monitored by a server program for incoming client requests. The endpoint is a network-specific address of a server

process for remote procedure calls. The name of the endpoint depends on the protocol sequence being used.

Endpoint Mapper (EPM): Part of the RPC subsystem that resolves dynamic endpoints in response to client requests and, in some configurations, dynamically assigns endpoints to servers.

Client Stub: Module within a client application containing all of the functions necessary for the client to make remote procedure calls using the model of a traditional function call in a standalone application. The client stub is responsible for invoking the organizing engine and some of the RPC application programming interfaces (APIs).

Server Stub : Module within a server application or service that contains all of the functions necessary for the server to handle remote requests using local procedure calls.

RPC Dependencies and Interactions: RPC is a client/server technology in the most generic sense. There is a sender and a receiver; data is transferred between them. This can be classic client/server (for example, Microsoft Outlook communicating with a server running Microsoft Exchange Server) or system services within the computer communicating with each other.

Message Passing vs. Remote Procedure Call

Message Passing and RPC are two major used methods of computer communication. RPC is built on top of MP. MP on one hand provides a non-reliable, asynchronous method of communication, RPC on the other hand, provides reliable and synchronous communication. The arguments between MP and RPC are going to be very similar to the arguments between UDP and TCP. Since, RPC is a layer on top of MP; End-to-End argument applies here. RPC fits better in a client server model while MP fits better in a peer-to-peer/Ad-hoc model of communication.

Alternative steps a programmer needs to take while using MP:

- Since reliability is not provided by default in MP, we can use reliable send (a send which will not return unless the message has been delivered to the other end) can be

used.

- Similarly, a reliable send would also provide us with synchronous behavior.
- Programmer can write his own marshalling and de-marshalling algorithms.
- Programmer can develop his own resource discovery mechanisms.

How does RPC provides these things:

- RPC does provide reliability
- RPC calls are synchronous by nature/implementation
- RPC abstracts away the marshalling/de-marshalling details from the programmer. Hence programmer doesn't have to worry about the same
- RPC provides central resource manager

To sum it up, RPC does provide extra functionalities but if they are not required in a certain scenario, these extra functionalities would end up creating lot delays and performance overheads, and hence MP would be a better option. But if you need to pass around complicated data structures and the scenario requires lot of synchronous communications among servers, then RPC would be a preferred choice.

RPC vs RMI

RPC (Remote Procedure Call) and RMI (Remote Method Invocation) are two mechanisms that allow the user to invoke or call processes that will run on a different computer from the one the user is using. The main difference between the two is the approach or paradigm used. RMI uses an object oriented paradigm where the user needs to know the object and the method of the object he needs to invoke. In comparison, RPC isn't object oriented and doesn't deal with objects. Rather, it calls specific subroutines that are already established.

RPC is a relatively old protocol that is based on the C language, thus inheriting its paradigm. With RPC, you get a procedure call that looks pretty much like a local call. RPC handles the complexities involved with passing the call from the local to the remote computer. RMI does the very same thing; handling the complexities of passing along the invocation from the local to the remote computer. But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called. RMI was developed by Java and uses its virtual machine. Its

use is therefore exclusive to Java applications for calling methods on remote computers.

In the end, RPC and RMI are just two means of achieving the same exact thing. It all comes down to what language you are using and which paradigm you are used to. Using the object oriented RMI is the better approach between the two, especially with larger programs as it provides a cleaner code that is easier to track down once something goes wrong. Use of RPC is still widely accepted, especially when any of the alternative remote procedural protocols are not an option.

1. RMI is object oriented while RPC isn't
2. RPC is C bases while RMI is Java only
3. RMI invokes methods while RPC invokes functions
4. RPC is antiquated while RMI is the future

III.3 Distributed Shared Memory

Distributed Shared Memory (DSM). DSM increases the complexity of the operating system but makes the job of application programmers far easier by allowing them to use the concept of shared memory when writing programs. Distributed shared memory is memory which, although distributed over a network of autonomous computers, gives the appearance of being centralized. The memory is accessed through virtual addresses, thus processes are able to communicate by reading and modifying data which are directly addressable. DSM allows programmers to use shared memory style programming, which makes application programming considerably easier. Programmers are able to access complex data structures and are relieved of the concerns of message passing. However, message passing cannot be avoided altogether. The operating system has to send messages between machines with requests for memory not available locally and to make replicated memory consistent.

1. Syntax

The syntax used for DSM is the same as that of normal centralized memory multiprocessor systems.

read(*shared_variable*)

write(*data, shared_variable*)

The **read()** primitive requires the name of the variable to be read as its argument and the **write()** primitive requires the data and the name of the variable to which the data is to be written. The operating system locates the variable through its virtual address and, if necessary, moves the portion of memory containing the variable to the machine requiring it.

2. Semantics

There are several issues related to the semantics of DSM.

***Structure and granularity of the shared memory*:** These two issues are closely related. The memory can take the form of an unstructured linear array of words or the structured forms of objects, language types or an associative memory. The granularity relates to the size of the chunks(cut) of the shared data. A decision has to be made whether it should be fine or coarse grained and whether data should be shared at the bit, word, complex data structure or page level. A coarse grained solution, page-based distributed memory management, is an attempt to implement a virtual memory model where paging takes place over the network instead of to disk. It offers a model which is similar to the shared memory model and is familiar to programmers, with sequential consistency at the cost of performance. Finer grained models can lead to higher network traffic.

***Consistency*:** In the simplest implementation of shared memory a request for a nonlocal piece of data results in a trap, which causes the single copy of the data to be fetched. If a piece of data was required by more than one machine the data could be moved backwards and forwards between the machines. This is very similar to thrashing in virtual memory and has the effect of considerably lowering performance.

The problem of thrashing is overcome by allowing multiple copies of data on the distributed machines. The problem then becomes one of maintaining the consistency of the replicated data. The cache coherence protocols of tightly coupled multiprocessors are a well researched topic, however many of these protocols are thought to be unsuitable for distributed systems because the strict consistency models used cause too much network traffic. Consistency models determine the conditions under which memory updates will be propagated through the system. These models can be divided into those with and those without synchronization operations. The former include strict, sequential, causal, processor and PRAM consistency models, while models with

synchronization operations include weak, release and entry consistency models.

There is a weakening of the consistency models from strict to entry consistency. Weaker models reduce the amount of network traffic hence the performance of the system improves. Thus, weaker consistency models have been used in an attempt to achieve better performance in distributed systems. However, this makes the programming model more complicated and makes weaker consistency the concern of operating systems and language designers.

Synchronization: Shared data must be protected by synchronization primitives, semaphores, event counts, monitors or locks. There are three methods of managing synchronization. Firstly, it can be managed by a synchronization manager, as in the case of page-based systems, or secondly, it can be made the responsibility of the application programmer, using explicit synchronization primitives, as in the shared variable implementation. Finally, it can be made the responsibility of the system developer, as in object based implementations, with synchronization being implicit at application level.

Heterogeneity: Sharing data between heterogeneous machines is an important problem for distributed shared memory designers. Data shared at the page level is not typed, hence accommodating different data representations of different machines, languages or operating systems is a very difficult problem. The Mermaid approach is to only allow one type of data on an appropriately tagged page.

The overhead of converting the data might be too high to make DSM on a heterogeneous system worth implementing.

Scalability: One of the benefits of DSM systems mentioned in much of the literature is that they scale better than many tightly-coupled shared memory multiprocessors. However, scalability is limited by physical bottlenecks, e.g., buses in tightly-coupled multiprocessor systems and operations which require global information or distribute information globally, e.g. broadcast messages.

CHAPTER III: APPROPRIATE APPLICATIONS IN DISTRIBUTED SYSTEMS

III.1 MIDDLEWARE

III.1.1 History of Middleware

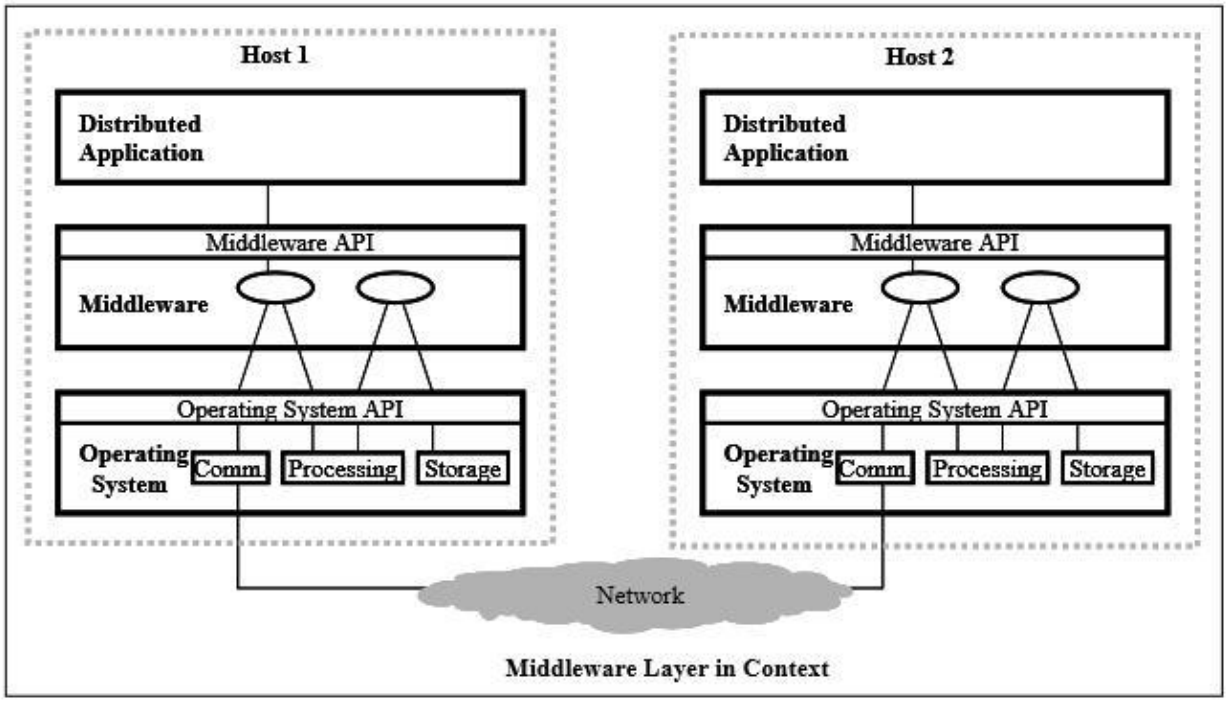
The term middleware first appeared in the late 1980s to describe network connection management software, but did not come into widespread use until the mid 1990s, when network technology had achieved sufficient penetration and visibility. By that time middleware had evolved into a much richer set of paradigms and services offered to help make it easier and more manageable to build distributed applications. The term was associated mainly with relational databases for many practitioners in the business world through the early 1990s, but by the mid-1990s this was no longer the case. Concepts similar to today's middleware previously went under the names of network operating systems, distributed operating systems and distributed computing environments.

Cronus was the major first distributed object middleware system (see Cronus), and Clouds (See Clouds) and Eden were contemporaries. RPC was first developed circa 1982 by Birrell and Nelson. Early RPC systems that achieved wide use include those by Sun in its Open Network Computing (ONC) and in Apollo’s Network Computing System (NCS). Open Software Foundation’s Distributed Computing Environment (DCE) included an RPC that was an adaptation of Apollo’s that was provided by Hewlett Packard (which acquired Apollo). Quality Objects (QuO) was the first middleware framework to provide general-purpose and extensible quality of service for distributed objects. TAO was the first major CORBA system to provide quality of service, namely real-time performance, directly in the ORB. The OMG was formed in 1989, and is presently the largest industry consortium of any kind. The Message Oriented Middleware Association (MOMA) was formed in 1993, and MOM became a widely-used kind of middleware by the late 1990s. In the late 1990s HTTP became a major building block for various kinds of middleware, due to its pervasive deployment and its ability to get through most firewalls.

III.1.2 Definition of Middleware

Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the

operating system but below the application program that provides a common programming abstraction across a distributed system. In doing so, it provides a higher-level building block for programmers than Application Programming Interfaces (APIs) such as sockets that are provided by the operating system. This significantly reduces the burden on application programmers by relieving them of this kind of tedious and error-prone programming. Middleware is sometimes informally called “plumbing” because it connects parts of a distributed application with data pipes and then passes data between them. Middleware frameworks are designed to mask some of the kinds of heterogeneity that programmers of distributed systems must deal with. They always mask heterogeneity of networks and hardware. Most middleware frameworks also mask heterogeneity of operating systems or programming languages, or both. A few such as CORBA also mask heterogeneity among vendor implementations of the same middleware standard. Finally, programming abstractions offered by middleware can provide transparency with respect to distribution in one or more of the following dimensions: location, concurrency, replication, failures, and mobility.



Middleware Layer in Context

The classical definition of an operating system is “the software that makes the hardware useable.” Similarly, middleware can be considered to be the software that makes a distributed system programmable. Just as a bare computer without an operating system could be

programmed with great difficulty, programming a distributed system is in general much more difficult without middleware, especially when heterogeneous operation is required. Likewise, it is possible to program an application with an assembler language or even machine code, but most programmers find it far more productive to use high-level languages for this purpose, and the resulting code is of course also portable.

III.1.2 Categories of Middleware

There are a small number of different kinds of middleware that have been developed. These vary in terms of the programming abstractions they provide and the kinds of heterogeneity they provide beyond network and hardware.

III.1.2.1 Distributed Tuples

A distributed relational databases offers the abstraction of distributed tuples, and are the most widely deployed kind of middleware today. Its Structured Query Language (SQL) allows programmers to manipulate sets of these tuples (a database) in an English-like language yet with intuitive semantics and rigorous mathematical foundations based on set theory and predicate calculus. Distributed relational databases also offer the abstraction of a transaction. Distributed relational database products typically offer heterogeneity across programming languages, but most do not offer much, if any, heterogeneity across vendor implementations. Transaction Processing Monitors (TPMs) are commonly used for end-to-end resource management of client queries, especially server-side process management and managing multi-database transactions. Linda is a framework offering a distributed tuple abstraction called Tuple Space (TS). Linda's API provides associative access to TS, but without any relational semantics. Linda offers spatial decoupling by allowing depositing and withdrawing processes to be unaware of each other's identities. It offers temporal decoupling by allowing them to have non-overlapping lifetimes. Jini is a Java framework for intelligent devices, especially in the home. Jini is built on top of Java Spaces, which is very closely related to Linda's TS.

III.1.2.2 Remote Procedure Call

Remote procedure call (RPC; see Remote Procedure Calls) middleware extends the procedure call interface familiar to virtually all programmers to offer the abstraction of being able to invoke a procedure whose body is across a network. RPC systems are usually synchronous, and thus

offer no potential for parallelism without using multiple threads, and they typically have limited exception handling facilities.

III.1.2.3 Message-Oriented Middleware

Message-Oriented Middleware (MOM) provides the abstraction of a message queue that can be accessed across a network. It is a generalization of the well-known operating system construct: the mailbox. It is very flexible in how it can be configured with the topology of programs that deposit and withdraw messages from a given queue. Many MOM products offer queues with persistence, replication, or real-time performance. MOM offers the same kind of spatial and temporal decoupling that Linda does.

III.1.2.4 Distributed Object Middleware

Distributed object middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques encapsulation, inheritance, and polymorphism available to the distributed application developer. The Common Object Request Broker Architecture (CORBA; see Common Object Request Broker Architecture) is a standard for distributed object computing. It is part of the Object Management Architecture (OMA), developed by the Object Management Group (OMG), and is the broadest distributed object middleware available in terms of scope. It encompasses not only CORBA's distributed object abstraction but also other elements of the OMA which address general purpose and vertical market components helpful for distributed application developers. CORBA offers heterogeneity across programming language and vendor implementations. CORBA (and the OMA) is considered by most experts to be the most advanced kind of middleware commercially available and the most faithful to classical object oriented programming principles. Its standards are publicly available and well defined. DCOM is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) and Component Object Model (COM). DCOM's distributed object abstraction is augmented by other Microsoft technologies, including Microsoft Transaction Server and Active Directory. DCOM provides heterogeneity across language but not across operating system or tool vendor. COM+ is the next-generation DCOM

that greatly simplifies the programming of DCOM. SOAP is a distributed object framework from Microsoft that is based on XML and HyperText Transfer Protocols (HTTP). Its specification is public, and it provides heterogeneity across both language and vendor. Microsoft's distributed object framework .NET also has heterogeneity across language and vendor among its stated goals.

Java has a facility called Remote Method Invocation (RMI) that is similar to the distributed object abstraction of CORBA and DCOM. RMI provides heterogeneity across operating system and Java vendor, but not across language.

However, supporting only Java allows closer integration with some of its features, which can ease programming and provide greater functionality. Marketplace Convergence of the Concepts .The categories of middleware above are blurred in the marketplace in a number of ways. Starting in the late 1990s, many products began to offer APIs for multiple abstractions, for example distributed objects and message queues, managed in part by a TPM. TPMs in turn often use RPC or MOM as an underlying transport while adding management and control facilities. Relational database vendors have been breaking the relational model and the strict separation of data and code by many extensions, including RPC-like stored procedures. To complicate matters further, Java is being used to program these stored procedures. Additionally, some MOM products offer transactions over multiple operations on a message queue. Finally, distributed object systems typically offer event services or channels which are similar to MOM in term of architecture, namely topology and data flow.

III.1.2.5 Middleware and Legacy Systems

Middleware is sometimes called a "glue" technology because it is often used to integrate legacy components. It is essential for migrating mainframe applications that were never designed to interoperate or be networked to service remote requests. Middleware is also very useful for wrapping network devices such as routers and mobile base stations to offer network integrators and maintainers a control API that provides interoperability at the highest level. Distributed object middleware is particularly well-suited for legacy integration, due to its generality. In short, it provides a very high

lowest common denominator of interoperability. CORBA, in particular, is typically used for this because it supports the most kinds of heterogeneity and thus allows the legacy components to be used as widely as possible. Programming with Middleware Programmers do not have to learn a new programming language to program middleware. Rather, they use an existing one they are familiar with, such as C++ or Java. There are three main ways in which middleware can be programmed with existing languages. The first is where the middleware system provides a library of functions to be called to utilize the middleware; distributed database systems and Linda do this. The second is through an external interface definition language (IDL; see Interface Definition Language). In this approach, the IDL file describes the interface to the remote component, and a mapping from the IDL to the programming language is used for the programmer to code to. The third way is for the language and runtime system to support distribution natively; for example, Java's Remote Method Invocation (RMI).

III.1.2.6 Middleware and Layering

There may be multiple layers of middleware present in a given system configuration. For example, lower-level middleware such as a virtually synchronous atomic broadcast service (see Virtual Synchrony) can be used directly by application programmers. However, sometimes it is used as a building block by higher-level middleware such as CORBA or Message-Oriented Middleware to provide fault tolerance or load balancing or both. Note that most of the implementation of a middleware system is at the "Application" Layer 7 in the OSI network reference architecture, though parts of it is also at the "Presentation" Layer 6 (see Network Protocols). Thus, the middleware is an "application" to the network protocols, which are in the operating system. The "application" from the middleware's perspective is above it.

III.1.2.7 Middleware and Resource Management

The abstractions offered by various middleware frameworks can be used to provide resource management in a distributed system at a higher level than is otherwise possible. This is because these abstractions can be designed to be rich enough to subsume the three kinds of low-level physical resources an operating system manages: communications, processing, and storage (memory and disks). Middleware's abstractions also are from an end-to-end perspective, not just that of a single host, which allows for a more global and complete view to a resource management system. All middleware programming abstractions by definition subsume

communications resources, but others vary in how well they incorporate processing and storage. cleanly integrate all three kinds of resource into a coherent package. This completeness helps distributed resource management but also makes it easier to provide different kinds of distributed transparencies such as mobility transparency.

III.1.2.8 Middleware and Quality of Service Management

Distributed systems are inherently very dynamic, which can make them difficult to program. Resource management is helpful, but is generally not enough for most distributed applications. Starting in the late 1990s, distributed systems research has begun to focus on providing comprehensive quality of service (QoS), an organizing concept referring to the behavioral properties of an object or system, to help manage the dynamic nature of distributed systems. The goal of this research is to capture the application's high-level QoS requirements and then translate them down to low-level resource managers. QoS can help runtime adaptively, something in the domain of classical distributed systems research. But it can also help the applications evolve over their lifetime to handle new requirements or to operate in new environments; issues more in the domain of software engineering but of crucial importance to users and maintainers of distributed systems. Middleware is particularly well-suited to provide QoS at an application program's level of abstraction. Also, the abstractions middleware systems offer can often be extended to include a QoS abstraction while still being a coherent abstraction understandable by and useful to the programmer. Distributed object middleware is particularly well-suited for this due to its generality in the resources it encapsulates and integrates. Providing QoS to applications can help them operate acceptably when usage patterns or available resources vary over a wide spectrum and with little predictability. This can help make the environment appear more predictable to the distributed application layer, and help the applications to adapt when this predictability is impossible to achive. QoS can also help applications be modifiable in a reasonable amount of time, because their assumptions about the environment are not hard-coded into their application logic, and thus make them less expensive to maintain. Middleware that includes QoS abstractions can enable these things by making an application's assumptions about QoS, such as usage patterns and required resources, explicit while still providing a high-level building block for programmers. Further, QoS enabled middleware is a high-level building block which shields distributed applications from the low-level protocols and APIs that ultimately provide QoS.

CHAPTER IV: BASIC CONCEPTS IN DISTRIBUTED SYSTEMS

IV.1 STANDARDS & PROTOCOLS

A distributed software application is one with two or more distinct components which can perform more or less autonomously (individually) and yet also interact together to perform an additional series of tasks together. These "parts" might reside on different machines, or different networks. The components that make up the WWW give us a classic example of a distributed application.

Just like human beings, software programs typically don't exist in isolation - rather they interoperate with other software programs to make up applications, and then applications interoperate with other applications to make up systems. And just as with human social process, software programs rely on certain guides that help to establish rules, patterns, and common understandings that help to govern their interaction.

This post introduces standards and protocols which are as important in creating a distributed software system as the actual software components themselves. Indeed software is defined in large part by different individual **programs** - logical instructions written in human readable languages and converted into machine code to effect an electronic process. But there is more to applications like the WWW, like your email system, or like MSN instant messenger than simply the software programs. While some of the rules of software can be defined by software programs, other rules which help to govern the interaction of those programs, can be pre-established and "pre-written" *outside* the actual programs as standards and protocols.

A **Standard** is a mutual understanding, a shared meaning that is typically derived from the adoption of well-established declarations (but can also be established through social conventions).

A **Protocol** is a mutually understand method of *operating*.

Standards and protocols establish rules which act as positive constraints on a system, providing a framework for that system, such that the components within it can operate along more strict and manageable interactions. They limit the surprises that the components in a software system has

to deal with. That may seem confusing at first but if you think about some real world examples you'll start to get it.Think of our roads and the system we have developed for driving. An example of a standard would be the regulation that says a street light should have three different colored lights, a certain hue of green, amber, and red. An example of a protocol would be how to respond when you encounter each of these scenarios. These help us to navigate the road systems in coordination. We also have examples of standards and protocols in civil society. Our constitution exists to establish standards, or guidelines, for managing the affairs of our country. Without some constraints to provide some structure our society might descend into chaos -and that's what would happen to a distributed software system if it had no framework to work within.

By the same token too much structure isn't good either. It is difficult to grow and adapt in a system with too many rules. Adding too much structure can even create a bias in the system especially when it is administered by a bureaucratic few.

IV.2 COUPLING

Coupling refers to the degree of direct knowledge that one class has of another. Coupling is the strength of interconnection between two software modules: the higher the strength of interconnection, the higher the coupling. This is not meant to be interpreted as encapsulation vs. non-encapsulation. It is not a reference to one class's knowledge of another class's attributes or implementation, but rather knowledge of that other class *itself*.

Strong coupling occurs when a dependent class contains a pointer directly to a concrete class which provides the required behavior. The dependency cannot be substituted, or its "signature" changed, without requiring a change to the dependent class. Loose coupling occurs when the dependent class contains a pointer only to an interface, which can then be implemented by one or many concrete classes. The dependent class's dependency is to a "contract" specified by the interface; a defined list of methods and/or properties that implementing classes must provide. Any class that implements the interface can thus satisfy the dependency of a dependent class without having to change the class. This allows for extensibility in software design; a new class implementing an interface can be written to replace a current dependency in some or all situations, without requiring a change to the dependent class; the new and old classes can be interchanged freely. Strong coupling does not allow this. An indication of the module inter-

connection strength (degree of interaction between modules or strength of dependencies between modules). Ranges from complete dependence to independence .Depends on

- References from one module to another
- Amount of data passed between them
- Amount of control one has over the other
- The complexity of the interface between them

IV.2.1 Measuring data element coupling

The degree of the loose coupling can be measured by noting the number of changes in data elements that could occur in the sending or receiving systems and determining if the computers would still continue communicating correctly. These changes include items such as:

1. adding new data elements to messages
2. changing the order of data elements
3. changing the names of data elements
4. changing the structures of data elements
5. omitting data elements

Methods for decreasing coupling

Loose coupling of interfaces can be dramatically enhanced when publishers of data transmit messages using a flexible file format such as XML or JSON to enable subscribers to publish clear definitions of how they subsequently use this data. For example, a subscriber could publish the collection of statements used to extract

information from a publisher's messages by sharing the relevant XPath expressions used for data transformation, or the JSON Schema. This would allow a responsible data publisher to test whether their subscriber's extraction methods would fail when a published format changes.

IV.2.2 Loose coupling

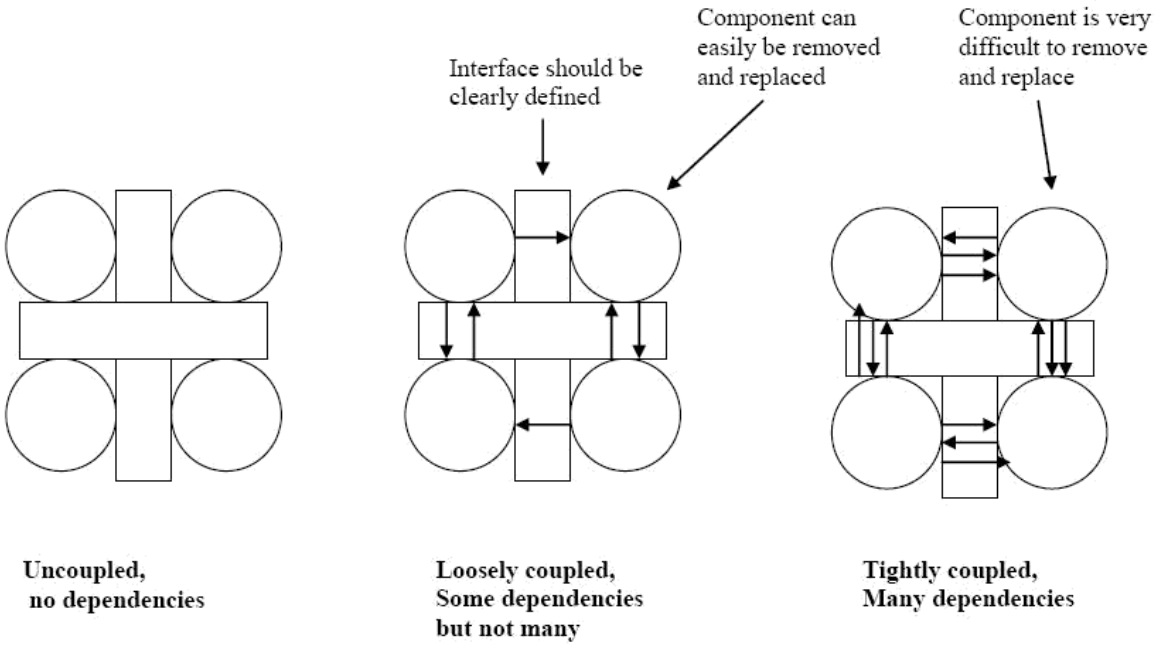
Loose coupling of services can be enhanced by reducing the information passed into a service to the key data. For example, a service that sends a letter is most reusable when just the customer

identifier is passed and the customer address is obtained within the service. This decouples services because services do not need to be called in a specific order (e.g. GetCustomerAddress, SendLetter)

Note that loose coupling is not universally positive. If systems are de-coupled in time using Message-oriented middleware, it is difficult to also provide transactional integrity. Data replication across different systems provides loose coupling (in availability), but creates issues in maintaining synchronisation.

IV.2.3 Tightly Coupled

A **tightly coupled system** is a multiprocessor computing system in which the CPUs are connected together in such a way that they share some or all of the system's memory and I/O resources.



Highly coupled designs give products that:

Are harder to understand, corrective maintenance is difficult (to determine where the fault is located), are difficult to extend, cannot be reused

Minimization of coupling

Allows consideration of modules in isolation
Makes it easier to understand how the module works, to test and modify the module.

High-availability clusters are groups of computers that support server applications that can be reliably utilized with a minimum of down-time. They operate by harnessing redundant computers in groups or clusters that provide continued service when system components fail. Without clustering, if a server running a particular application crashes, the application will be unavailable until the crashed server is fixed. HA clustering remedies this situation by detecting hardware/software faults, and immediately restarting the application on another system without requiring administrative intervention, a process known as failover. As part of this process, clustering software may configure the node before starting the application on it. For example, appropriate filesystems may need to be imported and mounted, network hardware may have to be configured, and some supporting applications may need to be running as well.

IV.3 Naming in Distributed System

- ***Entities, Names, Addresses***

An **Entity** in a distributed system can be pretty much anything.
A **Name** is a string of bits used to refer to an entity.
The **Address** is the name of the access point.

Identifiers are Special Names

• Can we use addresses of access points as regular name for the associated entity?
– access points may change over time
– entities may have several access points
• **Identifiers** uniquely identify an entity
– An identifier refers to at most one entity.
– Each entity is referred to at most one identifier.
– An identifier always referes to the same entity (never reused)

• Example: Telephone Numbers?

Name Space

Names are organized into **Name Space**. The set of names in a naming system. A naming space can be represented as a graph with leaf nodes (containing entity information) and directory nodes. Absolute and relative path names are related to a directory node A global name denotes the same entity in the system A local name depends on where the name is being used

Naming Systems

Unix/Linux file names

URL's on the World Wide Web

Types and objects in a C++ or Java program

Computers attached to the Internet

Types of Names in Distributed Systems Flat

All names are equivalent in name space

- Must be globally unique

Hierarchical

- Names (usually) have structure
- Unique only within immediately containing level
- Each level resolved within context of next higher level

Flat Name Spaces

Need global directory

- May be replicated
- May be partitioned

Not (necessarily) tied to location

- But many challenges Issues of scaling

Hierarchical Approaches

A flat name space with hierarchical administration

- Top level domain knows all names
- Each sub-domain knows subset of names
- Local names resolved within own subset
- Other names cached as needed

Domain Name System (DNS)

Domain Name System (**DNS**) is a hierarchical distributed naming system for computers, services, or any resource connected to the Internet or a private network It associates various information with domain name assigned to each of the participating entities. Most prominently, it translates easily memorized domain names to the numerical IP addresses needed for the purpose of locating computer services and devices worldwide. By providing a worldwide, distributed keyword -based redirection service, the Domain Name System is an essential component of the functionality of the Internet.

Internet names are structured, not flat

ccc3.wpi.edu ,**update.microsoft.com**

Resolution works the same way

- If a name is cached in local name server, try to use it
- If not, go to up the hierarchy one level to find a cached entry etc.
- Difference is that each level knows only its level
- E.g., **edu** knows **wpi** but not **ccc3**

IV.4 REPLICATION AND CONSISTENCY

IV.4.1 Consistency

In classical deductive logic, a consistent theory is one that does not contain a contradiction. The lack of contradiction can be defined in either semantic or syntactic terms. The semantic definition states that a theory is consistent if and only if it has a model, i.e. there exists an interpretation under which all formulas in the theory are true. A consistency proof is a mathematical proof that a particular theory is consistent.

In computer science, consistency models are used in distributed systems like distributed shared memory systems or distributed data stores (such as a file systems, databases, optimistic replication systems or Web caching). The system supports a given model if operations on memory follow specific rules. The data consistency model specifies a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of memory operations will be predictable.

IV.4.2 Replication

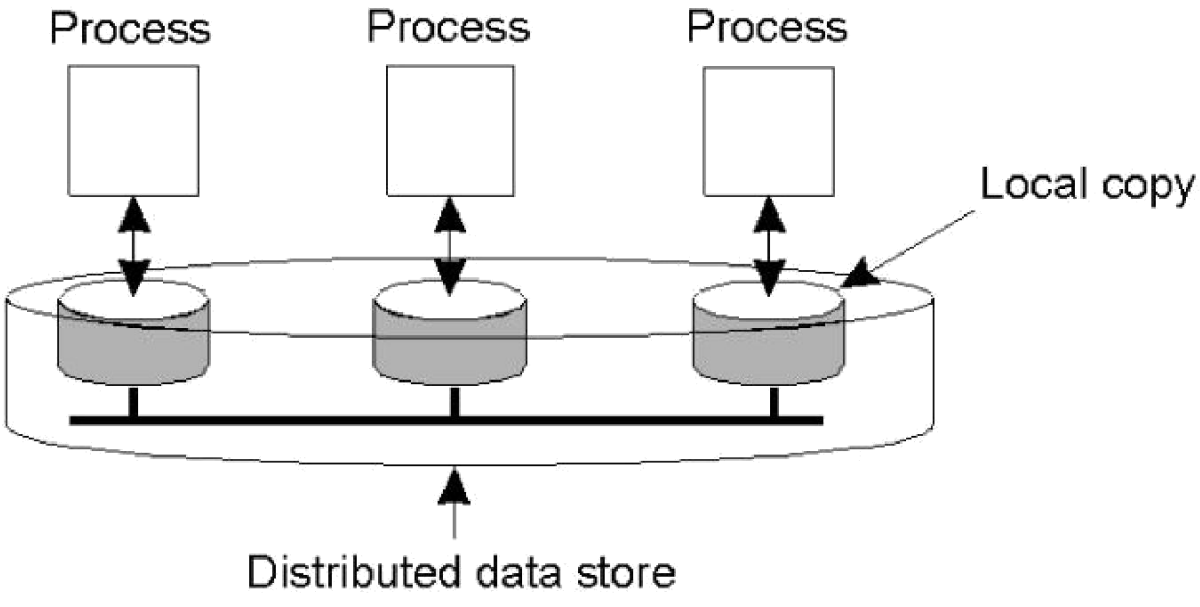
Replication in computing involves sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

Motivation Performance Enhancement, Enhanced availability, Fault tolerance, Scalability.

Tradeoff between benefits of replication and work required to keep replicas consistent.

Requirements: ***Consistency*** Depends upon application. In many applications, we want that different clients making (read/write) requests to different replicas of the same logical data item should not obtain different results. ***Replica transparency*** is desirable for most applications.

Data-Centric Consistency Models



(a)

The general organization of a logical data store, physically distributed and replicated across multiple processes.

Consistency Model is a contract between processes and a data store. If processes follow certain rules, then store will work correctly, Needed for understanding how concurrent reads and writes behave wrt shared data. Relevant for shared memory multiprocessors: cache coherence algorithms

Shared databases, files

- independent operations
- transactions

Strict Consistency

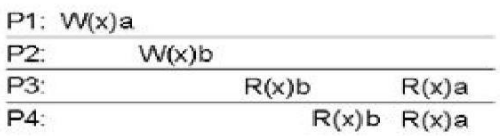
Any read on a data item x returns a value corresponding to the result of the most recent write on x.

A strictly consistent store

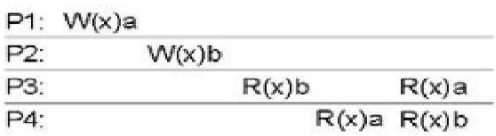
Behavior of two processes, operating on the same data item.

Sequential Consistency

Sequential consistency: the result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.



(a)



(b)

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.

Weak Consistency

Properties

- Accesses to synchronization variables associated with a data store are sequentially consistent
- No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
- No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

Release Consistency

Rules

- Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.
- Before a release is allowed to be performed, all previous reads and writes by the process must have completed
- Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).

Summary of Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

(a)

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

(b)

IV.5 FAULT TOLERANCE

IV.5.1 introduction

In the past, fault-tolerant computing was the exclusive domain of very specialized organizations such as telecom companies and financial institutions. With business-to-business transactions taking place over the Internet, however, we are interested not only in making sure that things work as intended, but also, when the inevitable failures do occur, that the damage is minimal.

Unfortunately, fault-tolerant computing is extremely hard, involving intricate algorithms for coping with the inherent complexity of the physical world. As it turns out, that world conspires against us and is constructed in such a way that, generally, it is simply not possible to devise absolutely foolproof, 100% reliable software. No matter how hard we

try, there is always a possibility that something can go wrong. The best we can do is to reduce the probability of failure to an "acceptable" level. Unfortunately, the more we strive to reduce this probability, the higher the cost.

There is much confusion about the terminology used with fault tolerance. For example, the terms "reliability" and "availability" are often used interchangeably, but do they always mean the same thing? What about "faults" and "errors"? In this section, we introduce the basic concepts behind fault tolerance.

Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. A complementary but separate approach to increasing dependability is fault prevention. This consists of techniques, such as inspection, whose intent is to eliminate the circumstances by which faults arise.

IV.5.2 Faults, Errors, and Failures

Implicit in the definition of fault tolerance is the assumption that there is a specification of what constitutes correct behavior. A failure occurs when an actual running system deviates from this specified behavior. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behavior specification. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures.

For example, in a software system, an incorrectly written instruction in a program may decrement an internal variable instead of incrementing it. Clearly, if this statement is executed, it will result in the incorrect value being written. If other program statements then use this value, the whole system will deviate from its desired behavior. In this case, the erroneous statement is the fault, the invalid value is the error, and the failure is the behavior that results from the error. Note that if the variable is never read after being written, no failure will occur. Or, if the invalid statement is never executed, the fault will not lead to an error. Thus, the mere presence of errors or faults does not necessarily imply system failure.

At the heart of all fault tolerance techniques is some form of masking redundancy. This means that components that are prone to defects are replicated in such a way that if a component fails,

one or more of the non-failed replicas will continue to provide service with no appreciable disruption. There are many variations on this basic theme.

IV.5.3 Reliability & Availability.

The two most common ways the industry expresses a system's ability to tolerate failure are reliability and availability.

Reliability is the likelihood that a system will remain operational (potentially despite failures) for the duration of a mission. For instance, the requirement might be stated as a 0.999999 availability for a 10-hour mission. In other words, the probability of failure during the mission may be at most 10^{-6}.Very high reliability is most important in critical applications such as the space shuttle or industrial control, in which failure could mean loss of life.

Availability expresses the fraction of time a system is operational. A 0.999999 availability means the system is not operational at most one hour in a million hours. It is important to note that a system with high availability may in fact fail. However, its recovery time and failure frequency must be small enough to achieve the desired availability. High availability is important in many applications, including airline reservations and telephone switching,in which every minute of downtime translates into significant revenue loss.

IV.5.4 Fault Classifications

Based on duration, faults can be classified as temporary or permanent. A transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed by some external agency. While it may seem that permanent faults are more severe, from an engineering perspective, they are much easier to diagnose and handle. A particularly problematic type of transient fault is the intermittent fault that recurs, often unpredictably.

A different way to classify faults is by their underlying cause. Design faults are the result of design failures, like our coding example above. While it may appear that in a carefully designed system all such faults should be eliminated through fault prevention, this is usually not realistic in practice. For this reason, many fault-tolerant systems are built with the assumption that design faults are inevitable, and theta mechanisms need to be put in place to protect the system against them. Operational faults, on the other hand, are faults that occur during the lifetime of the system

and are invariably due to physical causes, such as processor failures or disk crashes.

Finally, based on how a failed component behaves once it has failed, faults can be classified into the following categories:

- **Crash faults** -- the component either completely stops operating or never returns to a valid state;
- **Omission faults** -- the component completely fails to perform its service;
- **Timing faults** -- the component does not complete its service on time;
- **Byzantine faults** -- these are faults of an arbitrary nature.

IV.5.5 Failure Models In Distributed Systems

In this part of the paper, some failure models in distributed systems wikk be given. In all of these scenarios, clients use a collection of servers.

Crash: Server halts, but was working ok until then, e.g. O.S. failure.

Omission: Server fails to receive or respond or reply, e.g. server not listening or buffer overflow.

Timing: Server response time is outside its specification, client may give up.

Response: Incorrect response or incorrect processing due to control flow out of synchronization.

Arbitrary value (or Byzantine): Server behaving erratically (iregurally), for example providing arbitrary responses at arbitrary times. Server output is inappropriate but it is not easy to determine this to be incorrect. Duplicated message due to buffering problem maw be given as an example. Alternatively, there may be a malicious element involved.

After giving the concepts about the failure models, some of the examples about failure models are shown below:

Case: Client unable to locate server, e.g. server down, or server has changed.
Solution: Use an exception handler, but this is not always possible in the programming language

used.

Case: Client request to server is lost.

Solution: Use a timeout to await server reply, then re-send, but be careful about idempotent operations**.** If multiple requests appear to get lost assume ‘cannot locate server’ error.

Case: Server crash after receiving client request. Problem may be not being able to tell if request was carried out (e.g. client requests print page, server may stop before or after printing, before acknowledgement)

Solutions: Rebuild server and retry client request (assuming ‘at least once’ semantics for request). Give up and report request failure (assuming ‘at most once’ semantics) what is usually required is exactly once semantics, but this difficult to guarantee.

Case: Server reply to client is lost.

Solution: Client can simply set timer and if no reply in time assume server down, request lost or server crashed during processing request.

IV.6 Validation and verification

We originally proposed trace and replica-based validation for Web and application servers in Internet services .Trace-based validation is similar in flavor to fault diagnosis approaches that maintain statistical models of ``normal" component behavior and dynamically inspect the service execution for deviations from this behavior. These approaches typically focus on the data flow behavior across the systems components, whereas trace-based validation inspects the actual responses coming from components and can do so at various semantic levels.

Replica-based validation has been used before to tolerate Byzantine(complicated) failures and malicious attacks. In this context, replicas are a permanent part of the distributed system and validation is constantly performed via voting. Model-based validation is loosely related to two approaches to software debugging: model checking and assertion checking. Besides its focus on human mistakes, model-based validation differs from other assertion-checking efforts in that our

assertions are external to the component being validated. Model-based validation differs from model checking in that it validates components dynamically based on their behavior, rather than statically based on their source codes.

IV.7 Scheduling and Load Balancing

Scheduling and load balancing techniques are crucial for implementing efficient parallel and distributed applications and for making best use of parallel and distributed systems. These techniques can be provided either at the application level or at the system level. At the application level, the mapping of distributed and parallel applications on to infrastructures and the development of dynamic load balancing algorithms that are able to exploit the particular characteristics and the actual utilization of the underlying system are of particular relevance.

IV.8 Security

Security in simple terms means protection given to anything such as assets, information, property and human beings. The application of safeguards to protect data/keeping data safe

e.g. use of passwords to prevent access to data from accidental or malicious modification, destruction or damage or from unauthorised access.

Purpose of a Security

Confidentiality: data access is confined to those with specified authority to view the data.

Integrity: all system assets are operating correctly according to specification and in the way the current user believes them to be operating.

Availability: information is delivered to the right person, when it is needed.

- **Privacy**

Some data should only be accessed by authorised personnel have a responsibility not to disclose the private data to others. e.g. in a doctors surgery the doctors may have access to patients clinical data and general data but the receptionists will only have access to the general data

CHAPTER V. BASIC PROBLEMS AND CHALLENGES IN DISTRIBUTED SYSTEMS

V.1 TRANSPARENCY

Concealment of the separation of the components of a distributed system (single image view).

There are a number of forms of transparency

- Access: Local and remote resources accessed
- Location: Users unaware of location of resources
- Migration: Resources can migrate without name change
- Replication: Users unaware of existence of multiple copies
- Failure: Users unaware of the failure of individual components
- Concurrency: Users unaware of sharing resources with others

Is transparency always desirable? Is it always possible?

V.2 SCALABILITY

A system is said to be scalable if it can without suffering a noticeable loss of complexity . Scale has three dimensions:

Handle the addition of users and resources performance or increase in administrative

- Size: number of users and resources (problem: overloading)
- Geography: distance between users and resources (problem: communication)

- Administration: number of organizations that exert administrative control over parts of the system (problem: administrative mess)
- *Note:*
- Scalability often conflicts with (small system) performance
- Claim of scalability is often abused.

Techniques for scaling

- Decentralization
- Hiding communication latencies (asynchronous communication, reduce communication)
- Distribution (spreading data and control around)
- Replication (making copies of data and processes)

V.3 DECENTRALIZATION

Avoid centralizing

- Services (e.g., single server)
- Data (e.g., central directories)
- Algorithms (e.g., based on complete information).

With regards to algorithms:

- Do not require machine to hold complete system state
- Allow nodes to make decisions based on local info
- Algorithms must survive failure of nodes
- No assumption of a global clock

CHAPTER VI: DISTRIBUTED ALGORITHMS

A **distributed algorithm** is an algorithm, run on a distributed system, that does not assume the previous existence of a central coordinator.

VI .1 Election Algorithms

Many distributed algorithms such as mutual exclusion and deadlock detection require a **coordinator process**.When the coordinator process fails, the distributed group of processes must execute an **election algorithm** to determine a new coordinator process.These algorithms will assume that each active process has a unique **priority id.**

- The **coordinator election problem** is to choose a process from among a group of processes on different processors in a distributed system to act as the central coordinator.
- An **election algorithm** is an algorithm for solving the coordinator election problem. By the nature of the coordinator election problem, any election algorithm must be a distributed algorithm.

- A group of processes on different machines need to choose a coordinator
- A peer to peer communication: every process can send messages to every other process.
- Assume that processes have unique IDs, such that one is highest
- Assume that the priority of process P_i is i

VI .1.1 Bully Algorithm

At any moment, a process can receive an **election** message from one of its lower-numbered colleagues.The receiver sends an OK back to the sender and conducts its own election.Eventually only the bully process remains. The bully announces victory to all processes in the distributed group.

Background: any process P_i sends a message to the current coordinator; if no response in T time units, P_i tries to elect itself as leader. Details follow:

Algorithm for process P_i that detected the lack of coordinator

- Process P_i sends an "Election" message to every process with higher priority.
- If no other process responds, process P_i starts the coordinator code running and sends a message to all processes with lower priorities saying "Elected P_i"
- Else, P_i waits for *T'* time units to hear from the new coordinator, and if there is no response start from step (1) again.

Algorithm for other processes (also called P_i)

If P_i is not the coordinator then P_i may receive either of these messages from P_j

if Pi sends "Elected P_j"; [this message is only received if $i < j$]

Pi updates its records to say that P_j is the coordinator.

Else if P_j sends "election" message (i > j)

Pi sends a response to P_j saying it is alive

Pi starts an election.

VI .1.2 Election In A Ring => Ring Algorithm.

Assume that processes form a ring: each process only sends messages to the next process in the ring

3. Active list: its info on all other active processes

4. Assumption: message continues around the ring even if a process along the way has crashed.

Background: any process P_i sends a message to the current coordinator; if no response in *T* time units, P_i initiates an election

1. initialize active list to empty.
2. Send an "Elect(i)" message to the right. + add i to active list.

If a process receives an "Elect(j)" message

(a) this is the first message sent or seen
initialize its active list to [i,j]; send "Elect(i)" + send "Elect(j)"

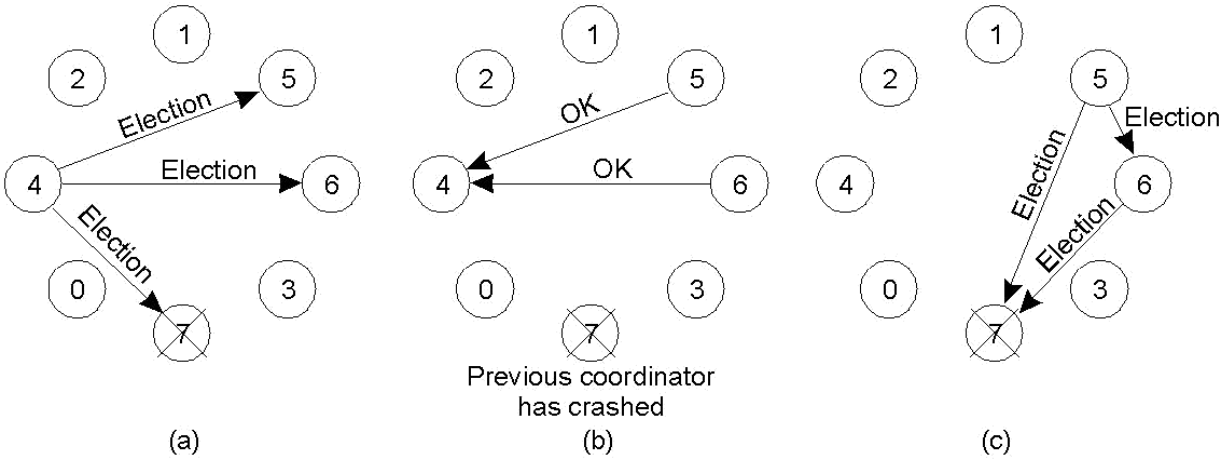
(b) if i != j, add i to active list + forward "Elect(j)" message to active list

(c) otherwise (i = j), so process i has complete set of active processes in its active list.

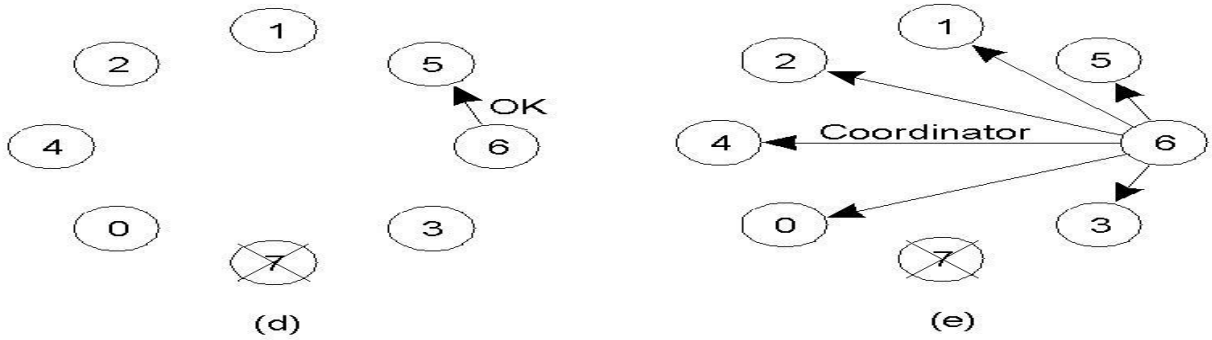
=> choose highest process ID + send "Elected (x)" message to neighbor

If a process receives "Elected(x)" message, set coordinator to x

Bully Algorithm Example:



- Process 4 notices 7 down.
- Process 4 holds an election.
- Process 5 and 6 respond, telling 4 to stop.
- Now 5 and 6 each hold an election.



- Process 6 tells process 5 to stop.
- Process 6 (the bully) wins and tells everyone.
- If processes 7 comes up, starts elections again.

A Ring Algorithm

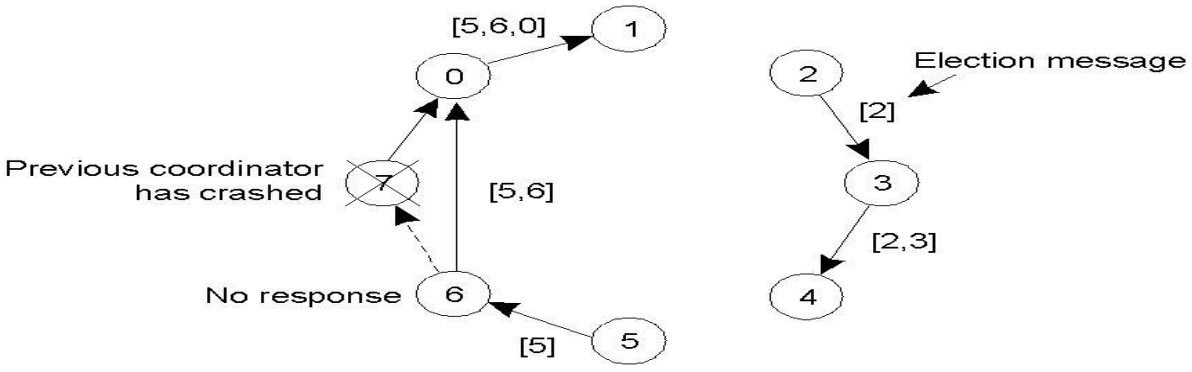
Assume the processes are logically ordered in a ring {implies a successor pointer and an active process list} that is unidirectional.

When any process, P, notices that the coordinator is no longer responding it initiates an election:

1.P sends message containing P's process id to the next available successor. At each active process, the receiving process adds its process number to the list of processes in the message and forwards it to its successor.

3. Eventually, the message gets back to the sender.

4. The initial sender sends out a second message letting everyone know who the coordinator is {the process with the highest number} and indicating the current members of the active list of processes.



- Even if two ELECTIONS start at once, everyone will pick the same leader.

VI.2 MUTUAL EXCLUSION

- To guarantee consistency among distributed processes that are accessing shared memory, it is necessary to provide mutual exclusion when accessing a critical section.
- Assume **n processes.**

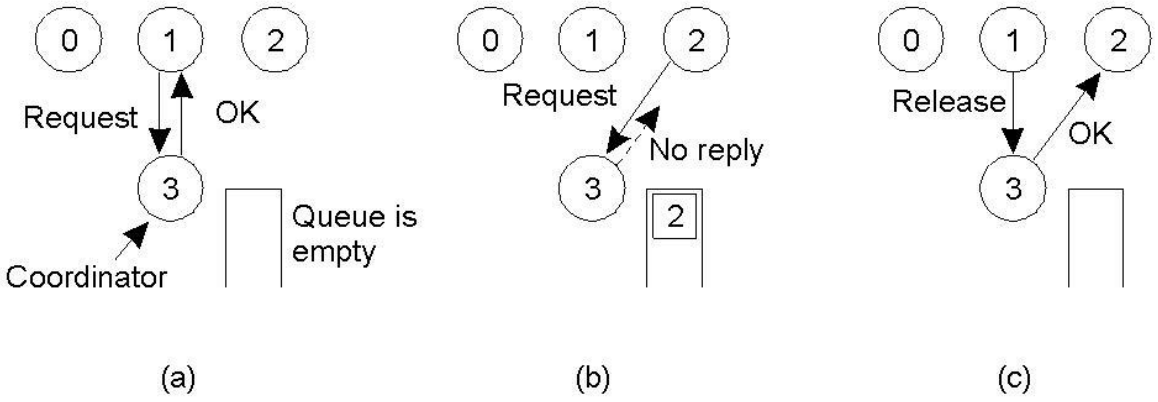
A Centralized Algorithm for Mutual Exclusion

Assume a coordinator has been elected.

- A process sends a message to the coordinator requesting permission to enter a critical section. If no other process is in the critical section, permission is granted.

- If another process then asks permission to enter the same critical region, the coordinator does not reply (Or, it sends "permission denied") and queues the request.
- When a process exits the critical section, it sends a message to the coordinator.

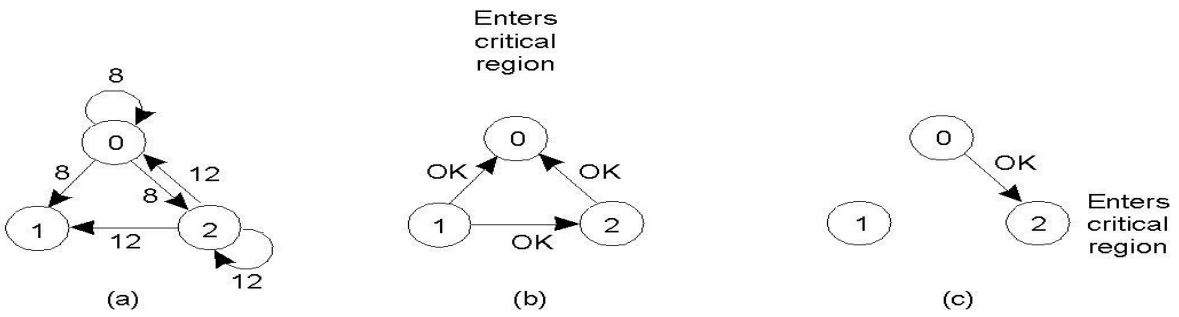
The coordinator takes first entry off the queue and sends that process a message granting permission to enter the critical section.



A Distributed Algorithm for Mutual Exclusion

Ricart and Agrawala algorithm (1981) assumes there is a mechanism for "totally ordering of all events" in the system (e.g. Lamport's algorithm) and a reliable message system.

1. A process wanting to enter critical sections (cs) sends a message with (cs name, process id, current time) to all processes (including itself).
2. When a process receives a cs request from another process, it reacts based on its current state with respect to the cs requested. There are three possible cases
3. If the receiver is not in the cs and it does not want to enter the cs, it sends an OK message to the sender.
4. If the receiver is in the cs, it does not reply and queues the request.
5. If the receiver wants to enter the cs but has not yet, it compares the timestamp of the incoming message with the timestamp of its message sent to everyone. {The lowest timestamp wins.} If the incoming timestamp is lower, the receiver sends an OK message to the sender. If its own timestamp is lower, the receiver queues the request and sends nothing.
6. After a process sends out a request to enter a cs, it waits for an OK from all the other processes. When all are received, it enters the cs.
7. Upon exiting cs, it sends OK messages to all processes on its queue for that cs and deletes them from the queue.



VI.3 FAULT-TOLERANT ALGORITHM

Paxos

Paxos is a popular fault-tolerant distributed consensus algorithm. It allows a globally consistant (total) order to be assigned to client messages (actions). Much of what is summarized here is from Lamport's *Paxos Made Simple* but I tried to simplify it substantially. Please refer to that paper for more detail and definitive explanations. The goal of a distributed consensus algorithm is to allow a set of computers to all agree on a single value that one of the nodes in the system proposed (as opposed to making up a random value). The challenge in doing this in a distributed system is that messages can be lost or machines cn fail. Paxos guarantees that a set of machines will chose a single proposed value as long as a majority of systems that participate in the algorithm are available. The setting for the algorithm is that of a collection of processes that can propose values. The algorithm has to ensure that a single one of those proposed values is chosen and all processes should learn that value.

There are three classes of agents:

1. Proposers
2. Acceptors
3. Learners

VI.3.1The Paxos algorithm

The Paxos algorithm operates in two phases:

Phase 1: Prepare: send a proposal request

Proposer:
A proposer chooses a proposal number *n* and sends a *prepare* request to a majority of acceptors. The number *n* is stored in the proposer's stable storage so that the proposer can ensure that a higher number is used for the next proposal (even if the proposer process restarts).

Acceptor:

- If an acceptor has received a proposal greater than *n* in the past, then it ignores this *prepare(n)* request.
- The acceptor promises never to accept a proposal numbered less than *n*.
- The acceptor replies to the proposer with a past proposal that it has accepted previously that had the highest number less than *n*: *reply(n',v').*

If a proposer receives the requested responses to its *prepare* request from a majority of the acceptors, then it can issue a proposal with number *n* and value *v*, where *v* is the value of the highest-numbered proposal among the responses or any value selected by the proposer if the responding acceptors reported no proposals.

Phase 2: Accept: send a proposal (and then propagate it to learners after acceptance)

Proposer:

A proposer can now issue its proposal. It will send a message to a set of acceptors stating that its proposal should be accepted (an *accept(n,v)* message). If the proposer receives a response to its *prepare(n)* requests from a majority of acceptors, it then sends an *accept(n, v)* request to each of those acceptors for a proposal numbered *n* with a value *v*, where *v* is the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

Acceptor:

If an acceptor receives an *accept(n, v)* request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than *n*.

The acceptor receives two types of requests from proposers: *prepare* and *accept* requests. Any request can be ignored. An acceptor only needs to remember the highest-numbered proposal that it has ever accepted and the number of the highest-numbered *prepare* request to which it has responded. The acceptor must store these values in stable storage so they can be preserved in case the acceptor fails and has to restart.

A proposer can make multiple proposals as long as it follows the algorithm for each one.

VI.4 CONSENSUS ALGORITHM

Now that the acceptors have a proposed value, we need a way to learn that a proposal has been accepted by a majority of acceptors. The *learner* is responsible for getting this information. Each acceptor, upon accepting a proposal, forwards it to all the learners. The problem with doing this is the potentially large number of duplicate messages:
*(number of acceptors) * (number of learners)*. If desired, this could be optimized. One or more *"distinguished learners"* could be elected. Acceptors will communicate to them
and they, in turn, will inform the other learners. *Consensus* is the task of getting all

processes in a group to agree on some specific value based on the votes of each processes. All processes must agree upon the same value and it must be a value that was submitted by at least one of the processes (i.e., the consensus algorithm cannot just invent a value). In the most basic case, the value may be binary (0 or 1), which will allow all processes to use it to make a decision on whether to do something or not.

With election algorithms, our goal was to pick a leader. With distributed transactions, we needed to get unanimous agreement on whether to commit. These are forms of consensus. With a consensus algorithm, we need to get unanimous agreement on some value. This is a simple-sounding problem but finds a surprisingly large amount of use in distributed systems. Any algorithm that relies on multiple processes maintaining common state relies on solving the consensus problem

Basic properties of Consensus

Termination: Every correct node eventually decides
Agreement: No two correct processes decide differently
Validity: Any value decided is a value proposed
Integrity: A node decides at most once

VI.5 TERMINATION DETECTION ALGORITHM

In practice, it is often necessary to know when the computation running in a distributed system has terminated. For example, it is possible to construct an efficient mutual exclusion algorithm in the following way: a first distributed algorithm establishes a spanning tree in the network, while a second algorithm circulates a token in a repeated depth-first traversal of the tree. To ensure the correctness of mutual exclusion, it is vital that the second algorithm is only started once the first algorithm has terminated, resulting in the problem of *termination detection*. A termination detection algorithm involves a computation of its own which should not interfere with the underlying computation which it observes. Additionally, it should satisfy two properties: (1) it should never announce termination unless the underlying computation has in fact terminated.

(2) If the underlying computation has terminated, the termination detection algorithm should eventually announce termination. But when is a computation in fact terminated? Answering this question means to define an appropriate formal notion of termination. To be general, the states of processes are mapped to just two distinct states: active and passive.An active process still actively participates in the computation while a passive process does not participate anymore unless it is activated by an active process. Activation can only be done using communication. For message-passing communication, which we also assume in this paper, a widely accepted definition of termination is that (1) all processes are passive and (2) all channels are empty.

Problems in the crash-recovery model

Solving the termination detection problem in the crash-recovery model is not an easy task. First of all, it is not clear what a sensible definition of termination is in the crash-recovery model. On the one hand, the classical (fault-free) definition of termination as mentioned above is clearly not suitable: If an active process crashes, there is always the possibility that it recovers later but there is no guarantee that it actually will recover. So an algorithm is in the dilemma to either making a false detection of termination or to possibly waiting infinitely long (see Figure). On the other hand, the definition used in the crash-stop model is also not suitable: An algorithm might announce termination prematurely if an active process which was crashed recovers again.

Failures

Crash-Recovery Model. We assume that processes fail by crashing and may recover subsequently. A process may fail and recover more than once. When a process *crashes*, it stops executing its algorithm and cannot send or receive messages. Processes have two types of storage: volatile and stable. If a process crashes, it looses the entire contents of its volatile storage. Stable storage is not affected by a crash. However, access to stable storage is expensive and should be avoided as much as possible.A *failure pattern* specifies times at which processes crash and (possibly) recover.It is a function F from the set of global clock ticks T to the power set of processes 2^{Π}. If $p \in F(t)$, then process p is crashed at time t. Given a failure pattern F, a process $p \in \Pi$

– is said to be *up at time* ***t,*** **if** $\boldsymbol{p \_ \in F(t)}$**.**

– is said to be *down at time* ***t,*** **if** $\boldsymbol{p \in F(t)}$**.**

– *crashes at time* $t \geq 1$ if p is up at time $t - 1$ and down at time t.

– *recovers at time* $t \geq 1$ if it is down at time $t - 1$ and up at time t.

As discussed in Aguilera *et al.* [1], a process p in the crash-recovery model

belongs to one of the four categories:

– ***Always-up:*** Process p never crashes.

– ***Eventually-up:*** Process p crashes at least once, but there is a time after which p is permanently up.

– ***Eventually-down:*** There is a time after which process p is permanently down.

– ***Unstable:*** Process p crashes and recovers infinitely many times.

Always-up and eventually-up processes are referred to as *good* processes. Eventually-down and unstable processes are referred to as *bad* processes. We use the phrases *up process* and *live process* synonymously. In this paper, we assume that communication channels among processes are *eventually-reliable*. A channel from process p to process q is said to be eventually reliable if it satisfies the following properties:

– ***Validity:*** If p sends a message to q and neither p nor q crashes, then the message is eventually delivered to q.

– ***No Duplication****:* No message is delivered more than once.

– ***No Creation:*** No message is delivered unless it was sent.

We also assume that all messages sent by a process are distinct. One way to ensure this is to maintain an *incarnation number* for a process in stable storage.

The incarnation number is incremented whenever the process recovers from a crash and written back to stable storage. In addition, there is a sequence number that is stored in volatile storage and is incremented whenever the process sends a message.

Each message is piggybacked with the incarnation number and the sequence number, which ensures that all messages sent by a process are distinct.

Failure Detectors in the Crash-Recovery Model

Many important problems in distributed computing such as consensus, atomic broadcast and termination detection are impossible to solve in an asynchronous distributed system when processes are unreliable [5]. To that end, Chandra and Toueg [3] introduced the notion of *failure detector*. A failure detector at a process outputs its current view about the operational state (up or down) of other processes in the system. Depending on the properties that failure detector output has to satisfy, several classes of failure detectors can be defined [3]. With the aid of failure detectors, problems such as consensus, atomic broadcast and termination detection become solvable in unreliable asynchronous distributed systems that are otherwise impossible to solve.

A failure detector itself is implemented by making certain synchrony assumptions about the system [3].The notion of failure detector, which was originally defined for crash-stop failure model, has been extended to crash-recovery failure model as well [1]. The failure detector defined by Aguilera *et al.* [1] for crash-recovery failure model, denoted by ♦*Se*, outputs a list of processes which are deemed to be currently up along with an epoch number for each such process. The epoch number associated with a process roughly counts the number of times the process has crashed and recovered. we use a failure detector with a simpler interface, denoted by ♦*Pcr*. The failure detector at a process only outputs a list of processes it currently deems to be up (we call this the *trust-list*). Processes which are not on the trust-list are suspected to be down by the failure detector. A suspected process is unsuspected if it is put on the trust-list. This failure detector satisfies the following properties:

– ***Completeness:*** Every *eventually-down* process is eventually permanently suspected by all good processes. Every *unstable* process is suspected and unsuspected
infinitely often by all good processes.
– ***Accuracy:*** Every good process is eventually permanently trusted by all good processes.

A failure detector from class ♦*P*cr is strictly stronger than a failure detector from class ♦*Se* because, in ♦*Se*, only one good process is required to be permanently trusted by all good processes. Nevertheless ♦*P*cr can be implemented under the same approach and the same assumptions of partial synchrony made in the original paper of Aguilera *et al.*[1].

The Termination Detection Problem

In the termination detection problem, the system is executing a distributed program, thereby generating a distributed computation referred to as *underlying computation*. A process in the system can be in two states with respect to the computation: *active* or *passive*. A process can execute an internal event of the computation only if it is active. There are three rules that describe allowed state changes between active and passive states:
– An active process may become passive at any time.
– A passive process can become active only on receiving a message.
– A process can send a message only when it is active.
We assume that processes have access to stable storage using which they are able to maintain their last saved state during time intervals when they are down.

Termination Detection Algorithm

Now that we know what it means for a computation to have terminated, we specify formally the properties of the termination detection problem:
– ***Liveness:*** If the underlying computation satisfies the termination condition, then the termination detection algorithm must announce termination eventually.
– ***Safety:*** If the termination detection algorithm announces termination, then the underlying computation has indeed terminated.
– ***Non-Interference:***The termination detection algorithm must not influence the underlying computation. To avoid confusion, we refer to messages sent by the underlying computation as

application messages and messages sent by the termination detection algorithm as control messages.

Impossibility of Termination Detection and its Consequences.

We assume that the processes have access to failure detector modules which observe the occurrence of failures in the system. Failure detectors are defined as general functions of the failure pattern, including functions that may provide information about future failures. Of course, such failure detectors cannot be implemented in the real world. Delporte-Gallet *et al.* introduced the notion of *realistic* failure detector. A failure detector is called realistic if it cannot guess the future behavior of the processes. In this work, we restrict ourselves to *realistic failure detectors*. To determine, if an execution is *robust-restricted terminated* it may be necessary to decide whether a currently down process is temporarily-down or forever-down. The two kinds of processes only differ in their future behavior. As a result, we postulate that no realistic failure detector can distinguish between a temporarily-down process and a forever-down process.

Algorithm Ideas

The main idea of our termination detection algorithm is that every process saves information about all messages it sends and receives. If the knowledge of all processes is combined, then the total set of messages in transit may be computed.Messages which are in transit towards a crashed process are assumed to be lost.If such a message is delivered anyhow, then the corresponding channel may have been incorrectly assumed to be empty as a result of which termination may have announced prematurely. The termination announcement is then revoked. Finally,the properties of the *stabilizing* crash-recovery model guarantee that eventually erroneous announcements of termination will end.The second idea is that every process is responsible for its own state and the state of all its incoming channels.

If a process becomes passive and it believes all its incoming channels to be empty, then it proposes by using a broadcast primitive the announcement of termination. If a process has received such a termination announcement proposal from all live process, it announces termination. For the termination detection algorithm, we use a *best-effort* broadcast primitive .Informally, best-effort broadcast guarantees that a broadcast message is correctly delivered to all processes which are currently up and do not crash while executing the broadcast protocol. All

currently up processes agree on the delivered message. Of course, all messages are delivered only once and no message is created by the best-effort broadcast primitive.The whole system consists of (1) the underlying computation *C* which is observed with respect to termination, (2) the superimposed and crash-recovery tolerant termination detection algorithm *A* we develop now, and (3) a failure detector *D* of class ♦*P*cr. If a passive process delivers a message, then it executes an becoming active event. We assume that all events that are executed satisfy the following: the corresponding instructions are executed within one time unit (atomically).

VI.6 STABILIZING

We use the notion of stabilization at two places: first, to restrict our problem to *stabilizing* termination detection, and second, to restrict the failure model to *stabilizing* crash-recovery model. Both assumptions are necessary as we now show: (1) Termination detector in the crash-recovery model: Not solvable (Corollary8). (2) Termination detection in the *stabilizing* crash-recovery model: when a process crashes, a realistic failure cannot distinguish between whether the crash is temporary or permanent.

If it assumes that crash is temporary (but it actually is permanent), then termination is never announced and the liveness property is violated. On the other hand, if it assumes that crash is permanent (but it actually is temporary), termination is announced prematurely and the safety property is violated. (3) *Stabilizing* termination detection in the *stabilizing* crash-recovery model: we provide an algorithm in the next section. (4) *Stabilizing* termination detection in the crash-recovery model: Assume a computation which never terminates because at least one active process crashes and recovers infinitely often. As a result, the termination detection algorithm will never cease announcing termination erroneously, because it expects the unstable process to "stabilize"—that is, to eventually cease crashing/recovering.

Therefore, our approach to solve *stabilizing* termination detection in the *stabilizing* crash-recovery model is a reasonable approach. In the next section we solve stabilizing termination detection in the stabilizing crash-recovery model using a failure detector from class ♦*P*cr. The next theorem shows that this kind of failure detector is also necessary. Hence, the failure detector

♦*P*cr is the weakest one for solving stabilizing termination detection in the stabilizing crash-recovery model.

VI.6.1 Solving Stabilizing Termination Detection.

In this section, we develop an algorithm for solving the stabilizing termination detection problem in the stabilizing crash-recovery model. It turns out that since we are solving a weaker version of the termination detection problem in more restricted crash-recovery model, we can *weaken* some of the assumptions we made earlier. First, we assume that a process, on recovery, may start in active state only if it crashed in active state.

This assumption may be weakened to: a process, on recovery, may start in active or passive state irrespective of the state in which it crashed. The actual state on recovery depends on the the application-specific recovery mechanism including the extent to which the application utilizes stable storage to log its state during execution. Second, we assume that channel does not duplicate any message. This assumption may be weakened to: a channel may duplicate a message finite number of times. With this weakened assumption, our channel can easily implemented on top of a fair-lossy channel using Retransmissions and acknowledgement.

VI.6.2 Self-Stabilization

Self-stabilization is a different approach to fault tolerance it
considers transient (temporary) failures
it is more optimistic
If bad thing happen (safety is violated), the system will recover within a finite time, and will behave nicely afterwards.

VI.6.2.1 Definition

A system is self-stabilizing when, regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps.
System *S* is self-stabilizing with respect to predicate *P* that identifies the legitimate states, if:

Convergence

Starting from any arbitrary configuration, *S* is guaranteed to reach a configuration satisfying *P*, within a finite number of state transitions.

Closure

P is closed under the execution of *S*. That is, once in a legitimate state, it will stay in a legitimate state.