Group 5

Members;

Cornelius Kipkorir – COM/016/15
Benedict Tuiya – COM/1002/15
Sharon Obongo – COM/044/15
Hazen Koskei – COM/031/15
Stacy Rutto – COM/005/15
Tracy Magoma – COM/013/15
Namu Ephantus – COM/006/15
Pretorius Ndung'u – COM/1017/14

COM 419: Computer Systems Design

Question 5: *Key concepts in distributed systems*

There are three basic concepts in distributed systems
1. Concurrency of components
2. Lack of a global clock
3. Independent failure of components

1. Concurrency of components

Concurrency is the property of a system representing the fact that multiple activities are executed at the same time .Vin Roy defined concurrency as "A program having several independent activities each of which executes at its own pace"
In concurrency the said activities above may perform some kind of interaction among them
The concurrent execution these activities can take place in environments including single-core processors ,multi-core processors, multi processors or even in multiple machines as part of a distributed system

These machines are all tasked to providing mechanisms to control different flows of execution via coordination and synchronization but also while ensuring consistency
Use of concurrency is motivated by performance gains
As explained by Cantrell et al, there are three ways by which concurrent execution of an application can improve performance

a) **Reduce latency**

A unit of work is executed in a shorter time by subdivision into parts that can be executed concurrently

b) **Hide latency**

Multiple long running tasks are executed together by underlying system
It is effective when tasks are blocked because of external resources they must wait upon, such as disk or network input/output operations

c) **Increase throughput**

Through multiple execution of tasks concurrently, throughput is increased which in turn speeds up the independent sequential tasks which might not have been specifically designed for concurrency
Concurrency is an intrinsic property of any kind of distributed system
Processes running on different machines form a common system that executes code on multiple machines at the same time

Concurrency and parallelism

Concurrency is a conceptual property of a program while parallelism is a run time state
Concurrency of a program depends on programming language and the way it is coded while parallelism depends on the actual run time environment
Concurrent execution may be performed sequentially (in any order) or simultaneously

Concurrent programming vs parallel programming

Concurrent programming tackles concurrent and interleaving tasks and the resulting complexity due to non-deterministic control flow
Parallel programming favors deterministic control flow and mainly reaches for optimized throughput
Parallel programming is suitable for:

-specialized tasks e.g graphics processing unit
High performance computing
Parallel programming advantage is that it takes computer clusters and distributes sub –tasks to cluster nodes, thus speeds up complex computations
Examples of architectures supporting concurrency include;

- SISD-single instruction stream, single data stream
- MIMD-multiple instruction stream, multiple data stream
- SIMD-single instruction stream, multiple data stream

Models for programming concurrency

a) **Sequential programming**

It is a deterministic programming model whereby there is no concurrency at all but there is a total order of operations of a program in its strongest form
While in a weaker form, they make no guarantees to exact execution order to the programmer

b) **Declarative concurrency**

This favors implicit programming flow of computations
Control flow is not described directly but rather a result of computational logic of the program
It allows multiple executions

c) **Message –passing programming**

This model allows concurrent activities to communicate via messages, this then is the only allowed form of interaction between activities which are otherwise completely isolated

d) **Shared-state concurrency**

This allows multiple activities to access contented resources and states
The sharing of resources and states among activities requires dedicated mechanisms for synchronization of access and coordination between activities

e) **Synchronization and coordination as concurrency**

Synchronization is a mechanism that controls access on shared resources between multiple activities
It is useful most when multiple activities require access to resources that cannot be accessed simultaneously

A proper synchronization works by employing exclusiveness and ordered access to the resource
Coordination aims at orchestration of a collaborating activities
Both activities can either be implicit or explicit
Implicit means the act of hiding the synchronization as part of the operational semantics of the programming language
Explicit synchronization requires the programmer add explicit synchronization operations to the code

Tasks processes and threads

It has been a crucial requirement for operating systems to execute multiple tasks concurrently, this is often addressed by multitasking
Multitasking mechanism then manages an interleaved and alternating execution of tasks it is often complimented by multiprocessing in cases of presence of multiple c p u's
Multiprocessing allocates different tasks on available CPU's
A process is a heavyweight task unit and owns systems resources that are allocated from operating system
Threads are lightweight task units that belong to a certain process
Concurrency, programming languages and distributed systems
There is a strong relationship between concurrent programming, programming languages and distributed systems when building large architectures
Programming distributed systems is challenging as compared to regular programming
These challenges include;

- Fault tolerance
- Integration of distributed aspect
- Reliability

In distributed systems build up, programming languages of Java, C++ are considered to be the best because they are combined with middle ware frameworks
Middle ware systems provided distributable compensate for features missing at the core of the language

2. Lack of a global clock

In distributed systems there are as many clock as there are systems
These clocks are coordinated to keep them somewhat consistent but no one clock has the exact time, even if the clocks were somewhat in sync, the individual clocks on each component may run at a different rate leading o them being out of sync only after one clock cycle
Therefore time is only known within a given precision
At the frequent intervals, a clock may synchronize with a more trusted clock .However, the clocks are not precisely the same because of the time lapses due to transmission and execution
Therefore the act of absence of a global clock in distributed systems is mainly brought about by the presence of many clocks by which they run on different rates.

3. Independent failures of components

The components of a distributed system running on different computers can continue to execute independent of each other.
For e.g. the network can fail thus isolating clients and servers which continue to run. Or a server can crash and the client may still be up.

Failure Handling

Failures in distributed systems are partial - some components fail while others continue to function. Hence failure handling can be difficult. Techniques can be used to:

a) Detect failures

E.g. use checksums to detect corrupt data in a file/message. Other times failure can only be suspected (e.g. a remote server is down) but not detected and the challenge is to manage in the presence of such failures.

b) Mask Failures

Some failures that have been detected can be masked/hidden or made less severe. E.g. messages can be retransmitted when then fail to be acknowledged. This might not help if the network is severely congested and in this case even the retransmission may not get through before timeout. Another e.g. File data can be

written to a pair of disks so that if one is corrupted, the other may still be correct (redundancy to achieve fault-tolerance).

c) Tolerate Failures

Most of the services on the Internet exhibit failures and it is not practical to detect or mask all the possible kinds of failures. In such cases, clients can be designed to tolerate failures. E.g. a web browser cannot reach a web server it does not make the user wait forever. It gives a message indicating that the server is unreachable and the user can try later.

Recovery from failures

This involves the design of software so that the state of permanent data can be recovered or "rolled back" after a server has crashed. E.g. database servers have a transaction handling ability that enables them to roll back a transaction that was not completed.