Compare Load sharing to task assignment and Load balancing strategies for scheduling processes in a distributed system.

Answer:

Load sharing:

- Load sharing means one can split the traffic from a network to be transported by different routers (paths).
- That's exactly what Cisco does with MHSRP. The document on Configuring Multichassis Multilink PPP states that when it tells it to configure half of the hosts with one default gateway and the second half with the other.
- Load sharing is inherent to the forwarding process of a router to share the forwarding of traffic, if the routing table has multiple paths to a destination.
- If equal paths, the forwarding process will decide the manner of forwarding and forward packets based on the load-sharing algorithm used.
- This still bears the possibility of unbalanced forwarding.
- If there are unequal paths, the traffic is distributed inversely proportionally to the cost of the routes.
- That is, paths with lower costs (metrics) are assigned more traffic, and paths with higher costs are assigned less traffic.
- Load sharing is a term used when attempting to share some of the traffic across multiple links.
- A good example of load sharing is when two devices connect using two links of different speed. Let's say link one is 9Mbit/s, and the other is 3Mbit/s.
- For every three packets we send through the 9Mbit link, we would want to send one packet down the 3Mbit/s link.
- The result is that the 9Mbit/s link would send a higher proportion of traffic than the 3Mbit/s link.

Load Balancing:

- Load balancing is a concept that aims to make a network more efficient.
- Load balancing distributes traffic load evenly across a network with multiple-paths, in order to get optimal resource utilization, maximize throughput and minimize response time.
- Thus load-balancing will split the traffic down the configured paths equally towards the destination.

- E.g., with two 768 kpbs links and 800 kpbs traffic at any point, conceptually with load-balancing each path should have 400 kpbs worth of traffic.
- Load balancing is an attempt to process traffic evenly across a network with multiple links.
- The reason this term is less preferred than load sharing is because it is difficult to achieve perfect load balancing.
- With load balancing, if I looked at two traffic graphs, I'd expect to see two identical amounts of bandwidth being used on each path to the destination.
- Because of the different ways I can achieve load balancing it can be difficult to achieve true load balancing across each of the paths.
- Load balancing means distributing the traffic evenly and dynamically among different paths to avoid link congestion and saturation.
- This can be done on a packet-by-packet basis or per destination in a round-robin fashion.
- The packets sent by a host follow different paths to the same destination. All paths belong to all hosts.

Explain Code migration and the role of Mobile agents.

Answer:

- Code migration in distributed systems takes place in the form of process migration. Moving a running process to a different machine is a costly and intricate task. The need for migration is performance.
- The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.
- Example: While searching for information on the Web it is easy to implement a search query in the form of a small mobile program that moves from site to site which will achieve a linear speed-up compared to using just a single program instance.
- Support for code migration can also help improve performance by exploiting parallelism, and also provide flexibility.

Dynamically configuring a client:

- If code can move between different machines, it becomes possible to dynamically configure distributed systems.

- This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine.
- The advantage of this model of dynamically downloading client side software, is that clients need not have all the software preinstalled to talk to servers. Another advantage is that as long as interfaces are standardized, the client-server protocol and its implementation can be changed.
- The disadvantage is lack of security due to the downloads.

Code migration model:

- The different models for code migration use a particular framework. A process consists of three segments:
- The code segment is the part that contains the set of instructions that make up the program that is being executed.
- The resource segment contains references to external resources needed by the process, such as files, printers, devices, other processes, etc.
- The execution segment is used to store the current execution state of a process, consisting of private data, the stack, and the program counter.

Type of mobility:

- Weak mobility- The minimum need for code migration is to provide only weak mobility. A characteristic feature of weak mobility is that a transferred program is always started from its initial state. Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable
- Strong Mobility- In systems that support strong mobility the execution segment can be transferred. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off. Clearly, strong mobility is much more powerful than weak mobility, but also much harder to implement.

Types of process to resource binding

- Identifiers- The strongest binding is when a process refers to a resource by its identifier. The process only requires precisely the referenced resource.

- Value- A weaker form of process-to-resource binding is when only the value of a resource is needed. The execution of the process would not be affected if another resource would provide that same value.
- Type- The weakest form of binding is when a process indicates it needs only a resource of a specific type.

Type of resources:

- Unattached-Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated
- Fastened- Moving or copying a fastened resource may be possible, but only at relatively high costs.
- Fixed- Fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices.
- Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that are needed when migrating code.

Mobile Agents:

- A mobile agent is simply an agent having the capability to move between different machines. Mobile agents often require support for strong mobility. The requirement of strong mobility is needed as agents are autonomous and actively interact with their environment.
- Moving an agent to another machine can hardly be done without considering its execution state. D'Agents system is an example of mobile agents where the combination of agents and weak mobility is useful.
- The ability to collaborate with other agents or to move between different machines are system properties of agents

State the desirable features of the global scheduling algorithm.

Answer:

No a priori knowledge about the processes:

Scheduling algorithms that operate based on the information about the characteristics and resource requirements of the processes pose an extra burden on the users who must provide this information while submitting their processes for execution.

Dynamic in nature:

Process assignment decisions should be dynamic, I.e., be based on the current load of the system and not on some static policy.

It is recommended that the scheduling algorithm possess the flexibility to migrate a process more than once because the initial decision of placing a process on a particular node may have to be changed after some time to adapt to the new system load.

Quick decision making capability:

Heuristic methods requiring less computational efforts (and hence less time) while providing near-optimal results are preferable to exhaustive (optimal) solution methods.

Balanced system performance and scheduling overhead:

Algorithms that provide near-optimal system performance with a minimum of global state information (such as CPU load) gathering overhead are desirable.

This is because the overhead increases as the amount of global state information collected increases.

This is because the usefulness of that information is decreased due to both the aging of the information being gathered and the low scheduling frequency as a result of the cost of gathering and processing the extra information.

Stability:

Fruitless migration of processes, known as processor thrashing, must be prevented.

E.g. if nodes n1 and n2 observe that node n3 is idle and then offload a portion of their work to n3 without being aware of the offloading decision made by the other node.

Now if n3 becomes overloaded due to this it may again start transferring its processes to other nodes.

This is caused by scheduling decisions being made at each node independently of decisions made by other nodes.

Scalability:

A scheduling algorithm should scale well as the number of nodes increases. An algorithm that makes scheduling decisions by first inquiring the workload from all the nodes and then selecting the most lightly loaded node has poor scalability.

This will work fine only when there are few nodes in the system.

This is because the inquirer receives a flood of replies almost simultaneously, and the time required to process the reply messages for making a node selection is too long as the number of nodes (N) increases.

Fault tolerance:

A good scheduling algorithm should not be disabled by the crash of one or more nodes of the system.

Also, if the nodes are partitioned into two or more groups due to link failures, the algorithm should be capable of functioning properly for the nodes within a group.

Fairness of service:

Global scheduling policies that blindly attempt to balance the load on all the nodes of the system are not good from the point of view of fairness of service.

This is because in any load-balancing scheme, heavily loaded nodes will obtain all the benefits while lightly loaded nodes will suffer poorer response time than in a stand-alone configuration.

A fair strategy that improves response time of the former without unduly affecting the latter is desirable.

Hence load-balancing has to be replaced by the concept of load sharing, that is, a node will share some of its resources as long as its users are not significantly affected.

Examine the distinctions between processes and threads, and assess the differences in execution between user-level and kernel-level threads.

<u>Answer:</u>

Process	**Thread**
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.

If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because the OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segments and code segments with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.

Types of Threads

There are two types of threads, which are:

1. User Level Thread

As the name suggests, the user-level threads are only managed by users, and the kernel does not have its information.

These are faster, easy to create and manage.

The kernel takes all these threads as a single process and handles them as one process only.

The user-level threads are implemented by user-level libraries, not by the system calls.

2. Kernel-Level Thread

The kernel-level threads are handled by the Operating system and managed by its kernel. These threads are slower than user-level threads because context information is managed by the kernel. To create and implement a kernel-level thread, we need to make a system call.

Features of Thread

- Threads share data, memory, resources, files, etc., with their peer threads within a process.
- One system call is capable of creating more than one thread.
- Each thread has its own stack and register.
- Threads can directly communicate with each other as they share the same address space.
- Threads need to be synchronized in order to avoid unexpected scenarios.

Explain the difference between data centric and client centric consistency models. Explain one model of each.

Answer:

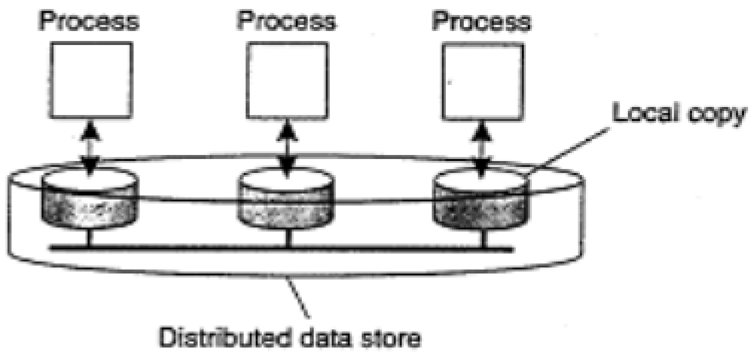
To reduce access time of data caching is used. The effect of replication and caching increases complexity and overhead of consistency management.

Different Consistency models offer different degrees of consistency.

Depending on the order of operations allowed and how much inconsistency can be tolerated, consistency models range from strong to weak.

Data centric model:

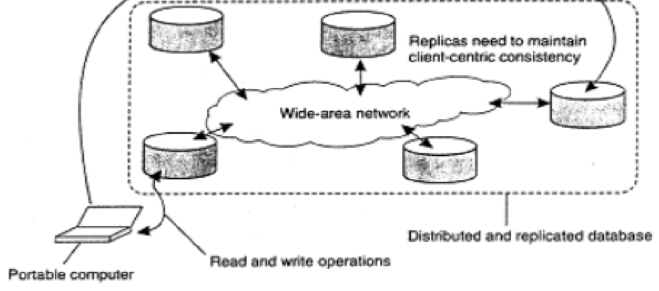
In this model it is guaranteed that the results of read and write operations which are performed on data can be replicated to various stores located nearby immediately.



Client centric model:

These consistency models do not handle simultaneous updates.

But, to maintain a consistent view for the individual client process to access different replicas from different locations has been carried out.


Client moves to other location and (transparently) connects to other replica
Replicas need to maintain client-centric consistency
Wide-area network
Distributed and replicated database
Read and write operations
Portable computer

Data centric consistency models explanation

Strict consistency:

It is the strongest data centric consistency model as it requires that a write on a data be immediately available to all replicas.

This model states that “Any read on data item x returns a value corresponding to the result of the most recent write on x.

In a distributed system, it is very difficult to maintain a global time ordering, along with the availability of data which is processed concurrently at different locations, which causes delays for transferring of data from one location to another.

Thus Strict consistency model is impossible to achieve.

Eg P1 & P2 are two processes. P1 performs a write operation on its data item x and modifies its value to a.

Update is propagated to all other replicas

Suppose if P2 now reads and finds its value to be NIL due to the propagation delay.

So the result of this value read by P2 is not strictly consistent.

But as the law of Strictly consistent model says that results should be immediately propagated to all the replicas as shown in figure below

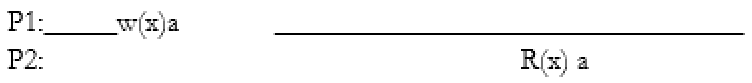


Fig (a) Strict consistent model

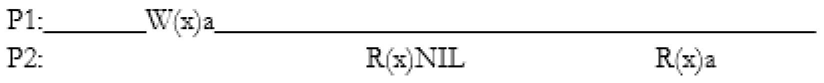


Fig (b) Propogation delay in strictly consistent model

Client centric model explanation

Monotonic Reads:

It states that “If a process P reads the value of a data item t, any successive read operation on x by that process at a later time will always return that same or a recent value.

For eg Each time one connects to the email server (may be different replicas),the email server guarantees that all the time will always return that same or recent value.

```
L1:____WS(x1) ______________________R(x1)________
 L2:     WS(x1,x2)                          R(x1)
```

Fig (a) Monotonic read consistent

Here, in fig (a) operations WS(x1) at L1 before the second read operation. Thus WS(x1,x2) at L2 would see the earlier update.

```
L1:__WS(x1) ___________________R(x1)_________
L2:   WS(x2)                        R(x2)
```

Fig (b) A data store which is monotonic read inconsistent

Justify how the Monotonic Read Consistency Model is different from Read Your Write Consistency Model. Support your answer with suitable examples.

Answer:

Client-Centric Consistency Models

Client-centric consistency models aim at providing a system wide view on a data store.

This model concentrates on consistency from the perspective of a single mobile client.

Client-centric consistency models are generally used for applications that lack simultaneous updates where most operations involve reading data.

Monotonic Reads Consistency

A data store is said to provide monotonic-read consistency if a process reads the value of a data item x, any successive read operation on x by that process will always return that same value or a more recent value.

A process has seen a value of x at time t, it will never see an older version of x at a later time.

Example: A user can read incoming mail while moving. Each time the user connects to a different email server, that server fetches all the updates from the server that the user previously visited. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

Read Your Writes

A data store is said to provide read-your-writes consistency if the effect of a write operation by a process on data item x will always be a successive read operation on x by the same process.

A write operation is always completed before a successive read operation by the same process no matter where that read operation takes place.

Example: Updating a Web page and guaranteeing that the Web browser shows the newest version instead of its cached copy.

Assess the significance of fault tolerance in distributed systems and analyze the role of recovery mechanisms in ensuring system reliability and availability.

Answer;

In distributed systems, there are three types of problems that occur. All these three types of problems are related.

Fault: Fault is defined as a weakness or shortcoming in the system or any hardware and software component. The presence of fault can lead to error and failure.

Errors: Errors are incorrect results due to the presence of faults.

Failure: Failure is the final outcome where the assigned goal is not achieved.

Fault Tolerance

Fault Tolerance is defined as the ability of the system to function properly even in the presence of any failure. Distributed systems consist of multiple components due to which there is a high risk of faults occurring. Due to the presence of faults, the overall performance may degrade.

Types of Faults

- Transient Faults: Transient Faults are the type of faults that occur once and then disappear. These types of faults do not harm the system to a great extent but are very difficult to find or locate. Processor fault is an example of transient fault.
- Intermittent Faults: Intermittent Faults are the type of faults that come again and again. Such as once the fault occurs it vanishes upon itself and then reappears again. An example of intermittent fault is when the working computer hangs up.
- Permanent Faults: Permanent Faults are the type of faults that remain in the system until the component is replaced by another. These types of faults can cause very severe damage to the system but are easy to identify. A burnt-out chip is an example of a permanent Fault.

Need for Fault Tolerance in Distributed Systems

Fault Tolerance is required in order to provide below four features.

- Availability: Availability is defined as the property where the system is readily available for its use at any time.
- Reliability: Reliability is defined as the property where the system can work continuously without any failure.
- Safety: Safety is defined as the property where the system can remain safe from unauthorized access even if any failure occurs.
- Maintainability: Maintainability is defined as the property that states how easily and fastly the failed node or system can be repaired.

Types of Fault Tolerance in Distributed Systems

- Hardware Fault Tolerance: Hardware Fault Tolerance involves keeping a backup plan for hardware devices such as memory, hard disk, CPU, and other hardware peripheral devices. Hardware Fault Tolerance is a type of fault tolerance that does not examine faults and runtime errors but can only provide hardware backup. The two different approaches that are used in Hardware Fault Tolerance are fault-masking and dynamic recovery.
- Software Fault Tolerance: Software Fault Tolerance is a type of fault tolerance where dedicated software is used in order to detect invalid output, runtime, and programming errors. Software Fault Tolerance makes use of static and dynamic methods for detecting and providing the solution. Software Fault Tolerance also consists of additional data points such as recovery rollback and checkpoints.
- System Fault Tolerance: System Fault Tolerance is a type of fault tolerance that consists of a whole system. It has the advantage that it not only stores the checkpoints but also the memory block, and program checkpoints and detects the errors in applications automatically. If the system encounters any type of fault or error it does provide the required mechanism for the solution. Thus system fault tolerance is reliable and efficient.

Recovery

Recovery from an error is essential to fault tolerance, and error is a component of a system that could result in failure. The whole idea of error recovery is to replace an erroneous state with an error-free state. Error recovery can be broadly divided into two categories.

1. Backward Recovery:

Moving the system from its current state back into a formerly accurate condition from an incorrect one is the main challenge in backward recovery. It will be required to accomplish this by periodically recording the system's state and restoring it when something goes wrong. A checkpoint is deemed to have been reached each time (part of) the system's current state is noted.

2. Forward Recovery:

Instead of returning the system to a previous, checkpointed state in this instance when it has entered an incorrect state, an effort is made to place the system in a correct new state from which it can continue to operate. The fundamental issue with forward error recovery techniques is that potential errors must be anticipated in advance. Only then is it possible to change those mistakes and transfer to a new state.

These two types of possible recoveries are done in fault tolerance in distributed systems.

Stable Storage :

Stable storage, which can resist anything but major disasters like floods and earthquakes, is another option. A pair of regular discs can be used to implement stable storage. Each block on drive 2 is a duplicate of the corresponding block on drive 1, with no differences. The block on drive 1 is updated and confirmed first whenever a block is updated. then the identical block on drive 2 is finished.

Suppose that the system crashes after drive 1 is updated but before the update on drive 2. Upon recovery, the disk can be compared with blocks. Since drive 1 is always updated before drive 2, the new block is copied from drive 1 to drive 2 whenever two comparable blocks differ, it is safe to believe that drive 1 is the correct one. Both drives will be identical once the recovery process is finished.

Another potential issue is a block's natural deterioration. A previously valid block may suddenly experience a checksum mistake without reason . The faulty block can be constructed from the corresponding block on the other drive when such an error is discovered.

Checkpointing :

Backward error recovery calls for the system to routinely save its state onto stable storage in a fault-tolerant distributed system. We need to take a distributed snapshot, often known as a consistent global state, in particular. If a process P has recorded the receipt of a message in a distributed snapshot, then there should also be a process Q that has recorded the sending of that message. It has to originate somewhere, after all.

Each process periodically saves its state to a locally accessible stable storage in backward error recovery techniques. We must create a stable global state from these local states in order to recover from a process or system failure. Recovery to the most current distributed snapshot, also known as a recovery line, is recommended in particular. In other words, as depicted in Fig., a recovery line represents the most recent stable cluster of checkpoints.