32. Given a link of the capacity of 10 Gbps, to send 3 flows with arrival rates of 3 Gbps, 6 Gbps, and 6 Gbps, compute the fair rates with the max-min fairness. If the weight of the first flow becomes 2 while the weights of the other flows remain as 1, compute their fair rates again.

Max-min fairness is a concept used in network routing to allocate resources as equally as possible amongst all users or data flows, without starving or giving an excess of resources to any one user or flow. The approach attempts to maximize the minimum allocation amongst all flows.

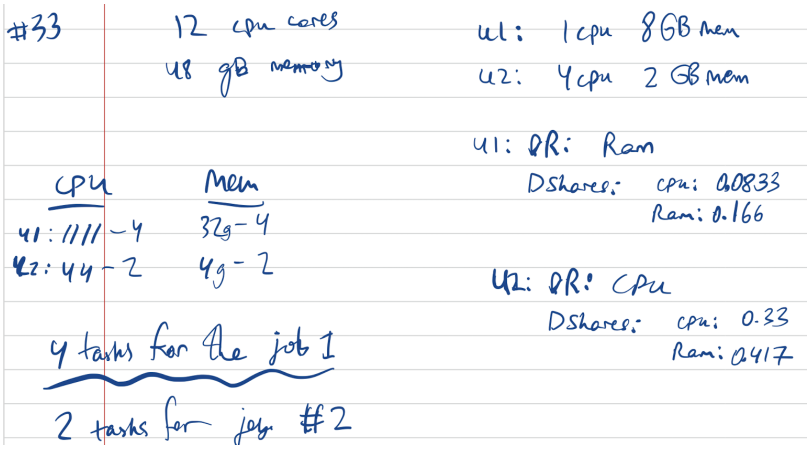
You have a link with a capacity of 10 Gbps.
You have three flows with arrival rates of 3 Gbps, 6 Gbps, and 6 Gbps.
The max-min fairness aims to provide the maximum possible minimum rate to all the flows. Here's how the algorithm works:

- First, you distribute the bandwidth equally amongst all flows, which means each gets 10/3 = 3.33 Gbps. But the first flow only needs 3 Gbps, so it gets that and the remaining 0.33 Gbps is split between the other two flows.
- Now, the first flow has 3 Gbps, and the other two have (3.33+0.165 = 3.495) Gbps each. Round up to 3.5.

If the weight of the first flow becomes 2 while the weights of the other flows remain as 1, we should distribute the rates again based on their weights. Here's how to do it:

- First, you divide the total link capacity by the total weight, which is 2 + 1 + 1 = 4. So each weight gets 10/4 = 2.5 Gbps.
- Then, you multiply each flow's weight by this per-weight rate. The first flow gets 2 * 2.5 = 5 Gbps, and the other two flows get 1 * 2.5 = 2.5 Gbps each.
- Now, the first flow has 5 Gbps, which is more than it needs, so it only takes 3 Gbps, and the remaining 2 Gbps is split equally between the other two flows (since their weights are equal).
- Therefore, the new fair rates are 3 Gbps for the first flow, and 3.5 Gbps for the second and third flows.

33. In a cluster of 12 CPU cores and 48 GB of memory, one job asks a resource scheduler to launch tasks of 1 CPU core and 8 GB of memory, and another job asks to launch tasks of 4 CPU cores and 2 GB of memory. What are the dominant resources of such two tasks? What are their dominant shares? With the dominant resource fairness, how many tasks can be launched in the cluster for the two jobs?



34. Why is the task with a finer granularity preferred by a cloud-scale resource scheduler?

The question you posed relates to resource scheduling in cloud-scale systems. In cloud computing, resource scheduling is the process of assigning and managing resources, such as computing power and storage, to different tasks or workloads.
In cloud-scale resource scheduling, tasks or workloads are typically divided into smaller units called tasks or subtasks. These tasks can range from simple computational tasks to complex jobs that require significant processing power and time.
A finer granularity refers to breaking down a task into smaller subtasks or smaller units of work. In cloud-scale resource scheduling, having tasks with finer granularity offers several advantages:

1. Improved Resource Utilization: Finer-grained tasks allow the resource scheduler to allocate resources more precisely based on the specific resource requirements of each subtask. This enables better resource utilization and reduces resource wastage.

2. Enhanced Load Balancing: With finer-grained tasks, the resource scheduler can distribute the workload more evenly across available resources. It can assign subtasks to different nodes or machines, balancing the computational load and avoiding resource bottlenecks.

3. Efficient Scaling: Fine-grained tasks facilitate efficient scaling of resources. The resource scheduler can dynamically adjust the allocation of resources based on the number and complexity of subtasks. This flexibility enables better responsiveness to changing workload demands and optimizes resource allocation.

4. Fault Tolerance: Finer-grained tasks improve fault tolerance in cloud-scale systems. If a subtask fails or encounters an issue, the resource scheduler can reroute or reschedule that specific subtask without impacting the entire workload. This isolation minimizes the impact of failures and improves system reliability.

In summary, finer granularity of tasks allows cloud-scale resource schedulers to optimize resource utilization, balance the workload, scale efficiently, and enhance fault tolerance. These advantages contribute to improved performance, better responsiveness, and overall efficient operation of cloud computing systems.

35. Compare the differences between the designs of Borg and Mesos.

The question you posed relates to the designs of two well-known cluster management systems: Borg and Mesos. These systems are designed to efficiently manage and allocate computing resources in large-scale distributed environments. Let's explore the background and then discuss the differences between Borg and Mesos.

Borg:

- Borg is an internal cluster management system developed by Google.
- It was designed to manage their vast data centers and handle the resource allocation and scheduling for various Google services and applications.
- Borg focuses on scalability, reliability, and efficient resource utilization. Some key characteristics of Borg include:
    - Centralized Control: Borg follows a centralized control model, where a central Borgmaster component oversees the allocation and management of resources across the cluster.
    - Job-centric Approach: Borg treats each application or workload as a job, which is further divided into tasks or processes. Borg schedules and assigns tasks to individual machines based on resource requirements and availability.
    - Resource Overcommitment: Borg allows resource overcommitment, meaning it can allocate resources beyond the available capacity by assuming that not all applications will demand their maximum resources simultaneously.
    - Tight Integration: Borg is closely integrated with other Google infrastructure components, such as the Google File System (GFS) for storage and Bigtable for data processing.

Mesos:

- Mesos is an open-source cluster management system initially developed at the University of California, Berkeley.
- It aims to provide a flexible and scalable platform for managing resources in distributed systems.
- Mesos focuses on resource sharing and isolation, fault tolerance, and support for different types of workloads.
- Key characteristics of Mesos include:
    - Distributed Architecture: Mesos follows a distributed architecture where multiple master nodes coordinate resource allocation and multiple slave nodes manage individual resources.
    - Resource Offers: Mesos uses the concept of resource offers, where slave nodes offer available resources to the master, and the master decides how to allocate those resources based on frameworks' requirements.
    - Framework Support: Mesos provides a framework API that allows different types of workloads, such as batch processing, stream processing, and containerized applications, to run on the Mesos cluster.
    - Fine-Grained Resource Allocation: Mesos supports fine-grained resource allocation, allowing tasks to request specific resource requirements, such as CPU and memory.

- Fault Tolerance: Mesos incorporates fault tolerance mechanisms to handle failures and ensure that tasks are rescheduled or migrated to healthy nodes in case of failures.

Now, let's compare the differences between Borg and Mesos:

- Origin and Openness: Borg is an internal system developed by Google and not openly available. On the other hand, Mesos is an open-source project, freely available for anyone to use and modify.
- Centralized vs. Distributed Control: Borg follows a centralized control model with a central Borgmaster, while Mesos employs a distributed control model with multiple master nodes.
- Resource Model: Borg treats applications as jobs and allocates resources to tasks, while Mesos operates with resource offers from slave nodes and allocates resources to frameworks.
- Fine-Grained Allocation: Mesos supports fine-grained resource allocation, allowing tasks to request specific resource requirements, while Borg focuses more on resource allocation for larger job-level units.
- Integration and Ecosystem: Borg is tightly integrated with Google's infrastructure components, while Mesos offers a more flexible framework API and supports various workload types, making it easier to integrate with different technologies and frameworks.
- Fault Tolerance: Both Borg and Mesos incorporate fault tolerance mechanisms, but the specific approaches and techniques may differ.

36. How does FaaS allow agile deployment and fast startup? (SEE P.71)

FaaS stands for Function as a Service, which is a type of cloud computing service that allows developers to execute and manage application functionalities without having to maintain the underlying infrastructure. The model is event-driven, meaning that the cloud provider runs the server and dynamically manages the allocation of machine resources.

Now, let's look at the question:

1. Agile Deployment: Agile deployment refers to a method of software deployment that prioritizes flexibility and speed. In FaaS, each function is packaged into a separate, stand-alone piece of code. These functions can be updated and deployed independently of one another, allowing for agile deployment. If a particular function needs to be changed or updated, you only need to modify that specific function rather than deploying a new version of the entire application. This allows for quicker, more responsive changes, reducing the risk associated with deployment and making it easier to roll back if there's a problem.

2. Fast Startup: FaaS is designed for applications that respond to events. Because of this, they are typically designed to start up quickly, in order to be ready when an event occurs. Also, cloud providers manage the server environment and can maintain "warm" instances of functions that are frequently used, reducing startup times even further. Additionally, FaaS eliminates the need for developers to write extensive setup or initialization code, allowing the services to start up and become operational quickly.

Therefore, FaaS enables agile deployment through its event-driven, function-based model, allowing individual components of an application to be updated and deployed independently. Furthermore, the managed nature of the server environment, along with the reduction in required setup code, allows for fast startup times.

37. Why stateful applications are bad cases for serverless computing (FaaS)? (SEE P.74)

Serverless computing, also known as Function as a Service (**FaaS**), provides a platform that allows developers to execute code in response to events without needing to provision or manage servers. This has its advantages such as cost efficiency, scalability, and reduced operational complexity.

However, serverless architectures are **stateless** - that is, they do not inherently maintain any knowledge of previous interactions. Each function execution is an independent, fresh computation without any retained state from previous computations. It's an inherent feature that enables easy scaling and high reliability, as each function can be run on any worker node at any time, independent of the others.

Now, let's delve into the question:

Stateful applications are applications that save client data generated in one session for use in the next session with that client. This could be any kind of data that needs to persist - like user profiles, e-commerce shopping carts, game scores, etc. They rely on the ability to read, write, and remember information, usually by tracking changes in data over time.

Stateful applications can be challenging for serverless computing because:

1. Serverless functions are stateless: As mentioned, serverless functions are designed to be stateless and independent. The absence of application state awareness can make it difficult to manage stateful applications where the state is important.

2. Cold starts: Serverless functions are subject to "cold starts" - if a function hasn't been invoked recently, it will need to be initialized before it can be run. This latency could be a problem for stateful applications that require frequent or rapid-fire access to their saved state.

3. Data sharing and synchronization: If the state needs to be shared across multiple functions or requires synchronization, this can be complex and challenging in a serverless architecture due to its distributed nature.

So while serverless computing can be incredibly efficient for stateless applications, it might not be the best choice for stateful applications unless managed properly. Strategies like using external databases, caching layers, or shared storage systems can help manage stateful applications in a serverless context but can also add complexity and potentially reduce some of the benefits of a serverless architecture.

38. Compare the differences between iterative and recursive queries. (SEE P.76)

In the context of key-value storage systems using a directory-based architecture, iterative and recursive queries can refer to how lookup requests are processed in the network.

1. Iterative Queries: In an iterative query, the client sends a request to the first server in the system (which could be any node in a distributed hash table, for example). If the server can't directly answer the request (because it doesn't have the key-value pair), it provides the client with an address of the next server to query. The client then directly contacts this next server with the same request. This process continues until the client finds the server containing the requested key-value pair. In other words, the client "iterates" over several servers until it finds the data it needs.

2. Recursive Queries: With recursive queries, the client sends a request to the first server. If this server can't directly answer the request, instead of sending back to the client the address of the next server to query, it forwards the client's request to this next server itself, and waits for a response. If this next server also can't directly answer the request, it forwards the request again, and so on, until the request reaches a server that can provide the requested data. The data then gets passed back up the chain to the first server, which finally sends it back to the client. Here, the lookup "recurses" through several servers until the data is found.

In terms of comparison:

- Iterative queries can be more efficient in terms of server resources, as each server is freed up as soon as it responds to the client. However, this could lead to more work for the client and possibly higher latency if many iterations are needed.

- Recursive queries may have lower latency, as the request is passed directly from server to server until the data is found, and the data takes the same route back. However, this could tie up resources on each server in the chain while it waits for responses from subsequent servers.

39. What is strong consistency? How does quorum consensus guarantee strong consistency when there is no node failure or network partition? (SEE P.78)

Strong consistency and quorum consensus are key concepts in distributed computing, especially in the context of distributed data stores. Here's an explanation:

1. Strong Consistency: Strong consistency is one of the consistency models used in distributed systems. Under strong consistency, any read operation that begins after a write operation completes will see the value written by that write operation. In other words, all operations appear to execute atomically and in a single, global order. Strong consistency ensures that all clients have the same view of the data at all times, making it appear as though there's just a single copy of the data.

2. Quorum Consensus: Quorum consensus is a method used to achieve strong consistency in distributed systems. A quorum is a minimum number of votes that a distributed transaction has to obtain in order to be allowed to perform an operation in a distributed system. The idea is to have the majority of nodes (a quorum) agree on a value before it's committed.

Quorum consensus guarantees strong consistency when there is no node failure or network partition in the following way:

In a write operation, the new value is written to a majority of the nodes (a write quorum) before the write is considered successful. Then, during a read operation, the value is read from a majority of the nodes (a read quorum). Since the write and read quorums overlap (they both include more than half of the nodes), any read operation that starts after a write operation completes will see the value written by that write operation, thus ensuring strong consistency.

In the event of a node failure, the overlapping quorums might not be achievable, potentially leading to inconsistent reads.

40. What is consistent hashing and why is it desirable for the distributed hash table? (SEE P.79)

Consistent hashing is a technique that is often used in distributed systems to distribute data across multiple nodes (machines). It's particularly useful in the context of distributed hash tables (DHTs), which are key-value stores distributed across multiple nodes.

Traditionally, when using a hash table in a distributed environment, if a node is added or removed from the system, nearly all keys will need to be remapped to different nodes because the range of the hash function has changed. This process is both computationally expensive and disruptive to the system as it causes a lot of data movement.

Consistent hashing is a scheme that largely minimizes this problem. It involves arranging the nodes of a distributed system in a hash ring or a circle. Each node is assigned a hash value on this circle. The keys are also hashed to a value on the circle, and each key is assigned to the first node that it encounters when moving clockwise on the ring from the point determined by its hash.

When a new node is added, it takes its place on the ring and takes over some of the keys from its neighbor. Similarly, when a node is removed, its keys are taken over by its neighbor. In both cases, only the keys in the range of the affected nodes need to be remapped, which is a minimal fraction of the total keys, especially in large systems.

This is why consistent hashing is desirable for distributed hash tables:

1. Minimizes reorganization of data: When nodes join or leave the network, only an average of K/n keys need to be remapped, where K is the total number of keys, and n is the number of nodes. This is in contrast to a traditional hashing scheme where potentially all keys could need to be remapped.

2. Balances load: Since keys are distributed across the ring, and each node handles keys that are closest to it on the ring, the load (i.e., number of keys) tends to be well-balanced across nodes.

3. High availability and fault tolerance: If a node fails, its keys are taken over by the next node in the ring. As long as the system can tolerate the extra load, this provides a high degree of availability and fault-tolerance.

41. How does the "finger table" in Chord save the number of hops to locate a key? (SEE P.80)

To answer this question, we first need to understand a few basics about the Chord protocol. **Chord** is a protocol and algorithm for a peer-to-peer distributed hash table (a DHT). A distributed hash table stores key-value pairs by assigning keys to different computers (or nodes); a key-value pair is stored on the node to which the key maps.

In the Chord protocol, each node maintains a routing table, also known as a finger table, which has up to "m" entries (where "m" is the number of bits in the keys), with the i-th entry pointing to the first node that succeeds the current node by at least 2^(i-1) on the identifier circle, where the successor of a node is the node with the next highest identifier.

The finger table enables fast lookups of keys. Instead of traversing the network node by node to find the correct location for a key, the finger table allows a node to skip directly to other nodes closer to the key.

If a key falls between a node and its i-th finger, then the key is guaranteed to be found in at most another i steps. This makes lookups very efficient. For a network of N nodes, any key can be found in O(log N) steps. Hence, the finger table in Chord drastically reduces the number of hops required to locate a key.

This logarithmic scaling property is a very desirable trait for large distributed systems, as it keeps search times relatively low even as the system scales to include many nodes.

42. How does hinted handoff work in Dynamo?

The question is in the context of Amazon's **Dynamo**, a highly available and scalable distributed data store built for Amazon's platform. Dynamo is a key-value storage system that is designed to provide highly reliable storage for relatively simple data models.

Hinted handoff is a technique used in Dynamo to handle temporary failures, such as when a node goes down for a short period of time. It's one of the techniques used to ensure that the system continues to operate effectively and that data is not lost when these types of problems occur.

The basic idea is this: when a write operation is performed, Dynamo attempts to store the data on N nodes (where N is a parameter of the system). If one of those nodes is down, Dynamo will store the data on another node and also include a "hint" indicating which node the data should have been stored on.

Later, when the failed node comes back online, the node that stored the "hinted" data will attempt to send that data back to the appropriate node. This process is called "handoff", because the data is being handed off from the node that temporarily stored it back to the node that should have stored it.

So, to sum up, the hinted handoff mechanism in Dynamo ensures the high availability and durability of the system even under server failures and network partitions.

43. What is eventual consistency? How does Dynamo maintain eventual consistency?

Eventual consistency is a consistency model used in many large distributed systems. It is a weaker form of consistency compared to others like strict consistency or linearizability, and it trades off immediate consistency for improved write performance and availability.

In an eventually consistent system, reads might not always reflect the results of a recently completed write. However, if no new updates are made to a particular item, eventually all accesses to that item will return the last updated value. In other words, the system eventually reaches a consistent state, hence the term "eventual consistency."

Dynamo is a distributed key-value storage system developed by Amazon that provides high availability at the cost of a weaker consistency model. It uses eventual consistency to provide high write availability, making it highly useful for write-intensive applications.

Dynamo maintains eventual consistency through several techniques:

1. **Conflict Resolution:** Dynamo allows for "write" operations to always succeed by employing a "last-write-wins" conflict resolution policy based on vector clocks. This allows the system to resolve conflicts without any intervention from the client.

2. **Replication and Consistency among Replicas:** When a write operation occurs, the data is stored across multiple nodes (replicas) in the system. Dynamo uses a quorum-like technique and only requires a majority of replicas to be written to before the write is considered successful.

3. **Hinted Handoff:** If a write operation fails to reach all of its designated replicas, Dynamo uses a technique called "hinted handoff" to increase consistency. The node that accepted the write but could not fulfill it to all replicas will keep a hint about the missed write operation and will try to fulfill it later when the unavailable replicas are up again.

4. **Anti-Entropy using Merkle Trees**:** Dynamo uses a mechanism called anti-entropy to keep replicas synchronized over time. It uses Merkle Trees for this purpose, which allows it to detect inconsistencies between replicas efficiently and repair them.

5. **Gossip-based Membership Protocol and Failure Detection:** Each node in Dynamo maintains a local view of the system's membership and its changes, and this information is propagated to other nodes using a gossip-based protocol. This helps in detecting node failures and maintaining eventual consistency.

44. List the limitations of a kernel function in CUDA.

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and API model created by NVIDIA. It allows developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing, an approach known as GPGPU (General-Purpose computing on Graphics Processing Units).

CUDA provides the concept of "kernels", which are functions that are written in a language like C or C++, but are executed N times in parallel by N different CUDA threads, unlike a traditional function that only executes once.

Despite their power and utility, kernel functions in CUDA have certain limitations:

1. **No support for recursive function calls:** CUDA kernels cannot call themselves, meaning recursion is not supported. This is due to the nature of GPU architecture, which doesn't have a stack to maintain function call hierarchies.

2. **Limited dynamic memory allocation:** While CUDA does support dynamic memory allocation, it can be slower and less efficient than using statically allocated memory. Additionally, memory allocated in one kernel cannot be freed in another.

3. **Limited use of system calls and standard library functions:** CUDA kernels cannot make system calls and can only use a subset of the standard library functions.

4. **No support for exception handling:** Unlike CPU programming, CUDA does not have built-in support for exceptions, so error handling mechanisms must be implemented manually.

5. **Synchronization limitations:** Synchronization among threads within a block is supported via the `__syncthreads()` function, but synchronization across blocks in a grid is not directly supported.

6. **Hardware limits:** The number of threads per block, the size of shared memory, and the number of registers are limited by the GPU hardware.

7. **Limited instruction set:** Only a subset of the instructions available on a CPU is available in a CUDA kernel.

These limitations require developers to take a different approach when writing code for the GPU, considering both the limitations and the strengths of the GPU's architecture.

45. Compare the differences between data parallelism and model parallelism.

Data Parallelism and Model Parallelism are techniques used to speed up training of machine learning models by distributing the computations across multiple processing units such as CPUs, GPUs, or machines in a network. Here's the background of each:

1. **Data Parallelism**: In this approach, the model parameters are replicated across every processing unit. Each unit then receives a unique subset of the total data and computes the model updates (gradients) for that subset. After all processing units finish computing their updates, they are typically averaged and the average update is applied to the model parameters. This is probably the most common form of parallelism in machine learning, because it's easier to implement and works well for many types of models, especially those with large amounts of data but models that can fit within the memory of a single processing unit.

2. **Model Parallelism**: In this approach, different parts of the model are placed on different processing units. For example, in a deep learning model, different layers might be placed on different GPUs. Each processing unit computes its part of the forward and backward pass, passing its results to the next processing unit. This approach is often used when a model is too large to fit on a single processing unit.

The main differences between these two are as follows:

- Data parallelism is typically easier to implement and more commonly used, especially in deep learning. However, it requires that the entire model can fit into the memory of each processing unit.

- Model parallelism can handle larger models that can't fit into the memory of a single processing unit, but it can be more difficult to implement efficiently due to the need to manage communication between processing units.

- Data parallelism can suffer from communication overhead when the model parameters are large and/or when there are many processing units, because the parameters (or updates to the parameters) need to be communicated between processing units. Model parallelism can suffer from communication overhead when different parts of the model need to communicate intermediate results frequently.

- In terms of applicability, data parallelism is generally suitable when you have large datasets and relatively smaller models, while model parallelism is suitable when your model size is too large to fit into a single processing unit's memory.

In practice, a combination of data parallelism and model parallelism is often used to leverage the advantages of both.

46. Compare the differences between synchronous and asynchronous training.

Synchronous and asynchronous training are both methods used in distributed machine learning to coordinate the updates to the model parameters when the training is distributed across multiple workers (machines or processing units).

Here's some background:

1. **Synchronous Training**: In this mode, all workers compute their updates to the model (based on their respective data subsets), then the updates are averaged and applied to the model. This happens in a synchronized manner, meaning all workers wait for each other to finish their computations before the updates are

applied. This method is simple and can lead to stable and consistent convergence because it is essentially equivalent to standard (non-distributed) training with a larger batch size. However, it can be slower because it requires all workers to wait for the slowest one in each iteration.

2. **Asynchronous Training**: In this mode, workers compute their updates and apply them to the model without waiting for the other workers. This can make the training process faster because it doesn't require waiting for the slowest worker. However, it can lead to problems with convergence because the model can change while a worker is still computing its update (this is known as a stale gradient problem).

The differences between the two methods, then, lie in their speed, their stability, and the complexity of their implementation:

- **Speed**: Asynchronous training can be faster because it doesn't require waiting for all workers to finish their computations. However, if the computations are not well-balanced across workers, some workers might be idle for long periods while waiting for others to catch up.

- **Stability and Convergence**: Synchronous training is more stable and can lead to more consistent convergence, but it can be slower. Asynchronous training can be faster, but it might lead to less consistent convergence due to the stale gradient problem.

- **Implementation Complexity**: Synchronous training is typically simpler to implement, while asynchronous training requires careful handling of the synchronization and potential conflicts between the workers.

In practice, a balance between the two methods is often sought. For example, a method called "elastic averaging" allows workers to update the model asynchronously most of the time, but periodically synchronizes the models across workers to prevent them from diverging too much.

47. Compare the communication overhead of single-master all-reduce, parameter-server all-reduce, and ring all-reduce.

The communication overhead in distributed machine learning is a critical factor to consider as it can substantially impact the training time and efficiency. Different communication strategies, such as single-master all-reduce, parameter-server all-reduce, and ring all-reduce, exhibit different behaviors concerning the communication overhead.

Here's some background:

1. **Single-Master All-Reduce**: In this strategy, one master node is responsible for gathering gradients from all worker nodes, averaging them, and then distributing the updated model back to the workers. This is a simple approach but it may put heavy network and computational burden on the master node, leading to a bottleneck if the number of worker nodes is large.

2. **Parameter-Server All-Reduce**: Here, a set of parameter servers are used to handle the gradients from the worker nodes. Each parameter server manages a portion of the model parameters. Workers send their computed gradients to parameter servers, which in turn update the respective parameters and send them back to the workers. While this can alleviate the single-master bottleneck, it might introduce additional network communication overhead due to the increased number of nodes involved.

3. **Ring All-Reduce**: In the ring all-reduce algorithm, nodes are arranged in a ring-like topology. Each node communicates only with its two neighboring nodes to pass around and aggregate gradients. After a full round of communication, all nodes have the averaged gradients. This method significantly reduces the communication overhead as it limits the number of communications. However, it may have longer latency due to the sequential nature of passing gradients around the ring.

Here's a comparison of the communication overhead:

- **Single-Master All-Reduce**: High communication overhead, especially for a large number of worker nodes, as all communication goes through a single master node.
- **Parameter-Server All-Reduce**: The communication overhead is distributed over several parameter servers, which can reduce the load on any single node, but might increase total network communication if the number of parameter servers is large.
- **Ring All-Reduce**: Low communication overhead as each node communicates only with its two neighbors. However, it may suffer from longer latency as gradient updates need to pass through all nodes sequentially.

In practice, the best method to use may depend on the specific characteristics of your network and the number of nodes involved in the computation.

48. Why is there over-subscription in the traditional data center network?

The traditional data center network architecture is hierarchical, typically designed in a tree structure, with the servers at the "leaves" and the routers or switches at the internal nodes. The communication bandwidth decreases as you move up the tree, and typically the network core (root of the tree) has the least bandwidth.

In the traditional model, the data center is typically oversubscribed. This means that the total potential demand for network bandwidth from servers (at the leaves of the tree) is more than the available bandwidth higher up in the tree. This is primarily due to cost and utilization considerations. Higher-speed networking equipment (routers, switches, cables, etc.) that could eliminate oversubscription is significantly more expensive. Furthermore, in many cases, not all servers need to communicate at full speed at the same time, so a certain level of oversubscription can be tolerated without noticeably impacting performance.

However, oversubscription can be a problem when many servers need to communicate with each other at the same time (for example, in "east-west" traffic patterns common in modern applications like MapReduce or virtual machine migration). In these situations, oversubscription can lead to network congestion and decreased performance. This has led to a shift towards flattened, non-hierarchical network architectures (such as Clos network or fat-tree architectures) that minimize oversubscription in modern data centers.

49. Briefly describe the features of FatTree and its benefits compared to the traditional topology.

FatTree is a specific type of network topology that is used within data centers to address the problem of oversubscription which is common in traditional hierarchical (tree-like) network topologies.

The FatTree architecture was proposed as a way to provide high-bandwidth, low-latency connectivity between all servers in a data center, regardless of their physical or logical location. In this architecture, the network is organized as a "fat tree" where each link in the network is provisioned with a bandwidth proportional to its position in the tree. This means that the higher up in the tree a link is, the more bandwidth it has.

The main features of FatTree are:

1. **Scalability:** FatTree is highly scalable and can support a large number of nodes (servers).

2. **High Bandwidth:** By allocating more bandwidth to the upper levels of the network, FatTree can prevent bottlenecks and provide high aggregate bandwidth across the data center.

3. **Fault Tolerance:** FatTree has built-in redundancy. If a link or switch fails, data can be rerouted along other paths.

4. **Load Balancing:** FatTree provides multiple paths between any two nodes in the network, allowing for better load balancing of network traffic.

Compared to traditional tree-like network topologies, FatTree offers several benefits:

1. **Elimination of Oversubscription:** In traditional tree-like topologies, the links at higher levels of the hierarchy are often oversubscribed, meaning they have less bandwidth than is potentially needed by the devices at the

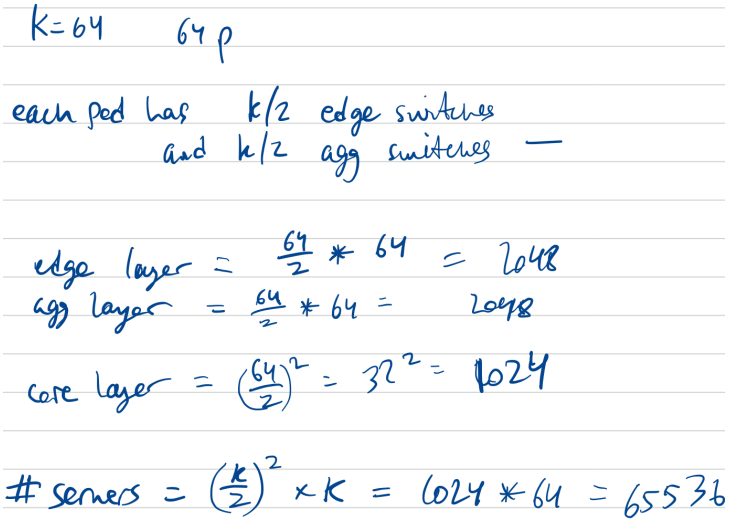
lower levels of the hierarchy. In FatTree, this is not an issue because the bandwidth of links increases as we go up the hierarchy.

2. **Increased Network Capacity:** FatTree offers full bisection bandwidth, which means any subset of servers can communicate with any other subset at the full bandwidth of their network interfaces, which is not possible in traditional hierarchical network designs.

3. **Improved Performance:** By eliminating bottlenecks and providing multiple paths between nodes, FatTree can deliver improved network performance, especially for data-intensive applications.

4. **Improved Flexibility:** FatTree does not have a rigid hierarchical structure like traditional topologies, giving it more flexibility in how nodes are connected and traffic is routed.

50. How many servers can be connected to a FatTree topology when k=64? How many switches are there in each layer? (SEE P.102)



51. Compare the differences between flow classification and flow scheduling?

In the context of networking and network management, flow classification and flow scheduling are two essential components that dictate how data is handled within a network.

Flow Classification is the process of categorizing network traffic based on a variety of factors such as source, destination, port, protocol, etc. This is often done using the packet headers which contain such information. The classification of the traffic can influence how it is handled by the network. For instance, network administrators might set rules that prioritize certain types of traffic or set policies that block or limit other types of traffic.

Flow Scheduling on the other hand is the process of managing when and how the classified flows are actually sent over the network. This includes deciding on the order of packet transmission, allocating bandwidth, and potentially even the route that each flow takes through the network. Scheduling decisions are often made based on the requirements of the flow (e.g., real-time video traffic may need to be sent immediately to prevent lag), the current network conditions, and the overall network policy.

In summary, flow classification is about identifying and categorizing network traffic, whereas flow scheduling is about managing the transmission of this traffic over the network. Both of these are essential for effective network management, especially in complex or congested networks where careful resource management is required.

52. Why is it difficult to deploy new network protocols on the Internet? How does SDN address such issues?

This question is addressing the challenges of introducing and deploying new network protocols on the Internet and how Software-Defined Networking (SDN) may provide solutions.

Network Protocols are a set of rules and conventions for communication between network devices. The Internet as we know it is governed by a stack of such protocols (like TCP/IP, HTTP, DNS, etc.) that have been developed and refined over decades. They are deeply entrenched in the infrastructure of the Internet, including in hardware, operating systems, and applications.

There are several reasons why deploying new network protocols on the Internet can be difficult:

1. **Compatibility**: The existing network infrastructure, which includes routers, switches, and servers, is designed to work with established protocols. Introducing new protocols may require updates or changes to existing hardware and software, which can be costly and time-consuming.

2. **Interoperability**: The Internet is a vast, decentralized network with a wide variety of devices and systems. Ensuring a new protocol works seamlessly across all these different systems can be a daunting task.

3. **Adoption**: Convincing network operators, service providers, and end-users to adopt a new protocol can be difficult, particularly if the benefits of the new protocol aren't immediately clear or if it requires significant changes to existing systems or practices.

Software-Defined Networking (SDN) is a technology that aims to address these issues. It separates the network's control (brains) and forwarding (muscle) planes to make network control directly programmable and the underlying infrastructure abstracted for applications and network services.

SDN helps to address the problems in the following ways:

1. **Flexibility**: Because the control plane is directly programmable, SDN networks can be more easily and quickly adapted to support new protocols.

2. **Centralized Control**: In an SDN network, a central controller has a holistic view of the network, which it can use to direct traffic and manage resources. This makes it easier to coordinate the introduction of new protocols across the network.

3. **Hardware Agnosticism**: The SDN abstraction allows for the network to be controlled independently from the physical hardware. Therefore, it's easier to implement new protocols without needing to make changes at the hardware level.

53. What is incast? How does DCTCP address the incast problem?

The "incast" problem and DCTCP (Data Center TCP) are both concepts related to data center networking.

Incast refers to a networking issue that can occur in data centers where many servers attempt to send data to a single receiver simultaneously. This simultaneous "many-to-one" communication can lead to network congestion, packet loss, and ultimately, a significant drop in application performance. It is a particularly common issue in data center applications like distributed storage systems and parallelized search systems, which often involve large numbers of servers responding to a single request.

The **DCTCP (Data Center TCP)** is a modification of the traditional TCP congestion control algorithm that was designed to better handle the specific challenges of data center networks, such as incast.

DCTCP addresses the incast problem in two ways:

1. **Explicit Congestion Notification (ECN)**: DCTCP uses Explicit Congestion Notification, a feature available in modern switches. When the switch buffer crosses a certain threshold, the switch marks the packet with a Congestion Experienced (CE) flag. This signal is used to indicate the start of congestion before packet loss actually occurs.

2. **Fine-grained Congestion Control**: Unlike traditional TCP, which responds to any indication of congestion by halving its sending rate, DCTCP adjusts its sending rate in a more fine-grained manner. If a small fraction of

packets are marked, it reduces its window by a small amount; if a large fraction are marked, it reduces the window by a larger amount.

By using these techniques, DCTCP can reduce the severity of the incast problem, achieving lower latency and better throughput in data center networks.

54. Why does MPTCP perform worse when the number of connections per host is too low or too high?

Multipath TCP (MPTCP) is an extension to the traditional TCP protocol that enables the simultaneous use of several IP-addresses/interfaces. This protocol is useful when resources are available along multiple paths, such as in data centers or mobile networks.

The purpose of MPTCP is to use multiple paths efficiently and reliably to maximize resource usage and increase redundancy. It's designed to operate transparently over most network infrastructures and application deployments.

Now, coming to the question:

- **Why does MPTCP perform worse when the number of connections per host is too low?**

When the number of connections is too low, there are fewer opportunities to take advantage of path diversity. With fewer connections, MPTCP has fewer options for splitting and routing data, and hence, it might not be able to fully utilize all available network resources.

- **Why does MPTCP perform worse when the number of connections per host is too high?**

When the number of connections is too high, there is an increased chance of congestion and potential collisions. This can lead to network instability and higher overhead due to the increased management of connections.

Moreover, each TCP connection (or subflow in the case of MPTCP) competes with others for bandwidth. If there are too many connections or subflows, they could lead to more competition and potential unfairness, resulting in degraded performance.

Thus, there is an optimal level of multiplicity in connections for achieving the best performance with MPTCP. Too few or too many can lead to suboptimal performance. It's a balancing act of utilizing multiple paths without overwhelming the network with too much competition for bandwidth.

55. Explain the concepts of spouts and bolts in Apache Storm, and provide examples of each.

Apache Storm is an open-source distributed real-time computation system for processing large volumes of high-velocity data. Apache Storm makes it easier and faster to process unbounded streams of data, doing real-time processing and distributed computation. It is often used for real-time analytics, online machine learning, continuous computation, distributed RPC, and ETL.

In the architecture of Storm, there are two types of components:

1. **Spouts**: A spout is a source of streams in a computation. Typically, it's designed to read data from an external source and emit tuples into the topology (the network of spouts and bolts). For example, a spout could connect to the Twitter API and continuously emit tweets into the topology for processing.

2. **Bolts**: A bolt processes any number of input streams and produces any number of new output streams. Most of the logic of a computation goes into bolts, such as functions, filters, streaming joins, streaming aggregations, talking to databases, and more. For instance, in the Twitter example, a bolt could process the tweet stream emitted by the spout and count the occurrence of each word.

So, in simple terms, a spout is a source of data, whereas a bolt consumes data and carries out processing or transformations. Together, they allow for flexible and powerful real-time data processing systems.

It's worth noting that a Storm topology, which consists of spouts and bolts, is a graph of stream transformations where each node is a spout or bolt, and edges represent the data flow between nodes. In a running topology, Storm manages the transmission of data from spouts to bolts and from bolts to other bolts.

56. Describe the role of the Nimbus and Supervisor daemons in the Apache Storm cluster.

Apache Storm operates on a master-worker architecture, with two types of daemons running on each node in the cluster: the Nimbus daemon and the Supervisor daemon.

Nimbus Daemon: The Nimbus daemon runs on a master node and is responsible for distributing code across the cluster, assigning tasks to machines, and monitoring for failures. In essence, the Nimbus daemon is the orchestrator of a Storm cluster, akin to the "JobTracker" in Hadoop's MapReduce. It doesn't process any data; instead, it ensures that everything else is operating correctly. If Nimbus fails, the system continues to process data, but no new tasks can be submitted or reassigned until Nimbus is restored.

Supervisor Daemon: The Supervisor daemon runs on worker nodes and listens for work assigned to its machine. It starts and stops worker processes as per the instructions from Nimbus. Each Supervisor node has multiple worker processes, and each worker process runs a specific topology and can execute multiple tasks across different threads. If a Supervisor node goes down, Nimbus will recognize the failure (because it periodically checks the status of the Supervisor) and reassign the tasks assigned to that Supervisor to other Supervisor nodes in the cluster.

In summary, Nimbus and Supervisor daemons form the backbone of Apache Storm's fault-tolerant, distributed computing capabilities. Nimbus ensures that all tasks are distributed correctly and monitored for failures, while Supervisors carry out the tasks and manage the worker processes.

57. Explain DStream in Spark Streaming and how it relates to RDDs in Spark.

Spark Streaming is an extension of the core Apache Spark API that allows real-time data processing. It is built on the powerful primitives provided by the Spark Core, particularly the Resilient Distributed Dataset (RDD), which is a fundamental data structure of Spark.

A Discretized Stream (DStream), is the basic abstraction provided by Spark Streaming. It is a sequence of data arriving over time. In Spark Streaming, incoming live data is divided into batches, and each batch of data is treated as RDDs, and then transformed into a new RDD. Hence, DStreams are essentially a sequence of RDDs, allowing Spark Streaming to seamlessly integrate with any other Spark components like MLlib and Spark SQL.

Each RDD in a DStream contains data from a certain interval, which is configurable. All transformations applied on a DStream translate to operations on the underlying RDDs. For instance, a map operation on a DStream leads to a map operation on each RDD in the DStream sequence. The processed results of the RDD operations are returned in batches as new DStreams.

In summary, DStream in Spark Streaming is a high-level abstraction that represents a continuous stream of data. DStreams can be created from various data sources, including Kafka, Hadoop HDFS, and more. Internally, each DStream is represented as a sequence of RDDs, which allows Spark Streaming to provide the same degree of fault tolerance, scalability, and ease of integration as batch processing in Spark.

58. Compare erasure coding with replication in data storage. Discuss the advantages and disadvantages of each approach in the context of distributed storage systems.

Erasure coding and replication are both methods used for data protection in distributed storage systems, but they work in fundamentally different ways and each comes with its own advantages and disadvantages.

Replication:
Replication works by storing multiple copies (usually 3) of the same data across different nodes in a distributed system. This approach is simple to implement and provides strong data durability and availability because even if one or two nodes fail, the data is still accessible from the other nodes.

However, replication has its downsides:
1. High storage overhead: Storing multiple copies of the same data requires significant storage resources.
2. Higher network bandwidth usage: Data replication involves moving multiple copies of the data across the network.

Erasure coding:
Erasure coding, on the other hand, is a method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces, and stored across a set of different locations or storage media.

The main advantage of erasure coding is that it significantly reduces the storage overhead compared to replication. Erasure coding schemes (like Reed-Solomon codes) can tolerate the same or higher number of failures as replication with much less storage overhead.

However, there are also downsides to erasure coding:
1. Computational overhead: The encoding and decoding process requires a significant amount of computational resources.
2. Higher latency: Retrieving and reconstructing erasure-coded data can lead to higher latency than replication.
3. Less efficient for small read/write operations: Erasure coding is less efficient than replication for small read/write operations because it requires reading/writing entire data stripes.

In general, the choice between replication and erasure coding in distributed storage systems depends on the specific requirements and constraints of the system. For systems that value simplicity, quick access, and are not as concerned with storage efficiency, replication might be the best choice. Conversely, for systems that need to store large amounts of data efficiently and can afford the computational overhead and potential increase in latency, erasure coding might be the better option.

59. When is reconstruction necessary in a distributed storage system?

Distributed storage systems store data across multiple nodes to ensure data availability and reliability. However, due to the distributed nature of these systems, nodes can fail or become temporarily inaccessible due to network issues, hardware failure, maintenance, etc. When such a failure occurs, the data stored on the failed node may become unavailable.

Reconstruction is necessary in a distributed storage system when a node fails and the data stored on that node needs to be restored or reconstructed on another node. This process is crucial to maintain the high data availability and fault tolerance of the system. Reconstruction typically involves reading the data from other nodes (that store either replicated copies or erasure-coded fragments of the data) and writing it to a new node.

60. In a Local Reconstruction Code, how does the local parity block reduce the reconstruction overhead of data blocks?

Local Reconstruction Codes (LRCs) are a type of erasure code used in distributed storage systems to provide high data durability while minimizing the cost of repairing lost data blocks (reconstruction overhead).

In LRCs, data is divided into blocks and additional parity blocks are calculated. These blocks are distributed across multiple nodes. Some of these parity blocks are "local" parities, calculated over a small subset of data blocks. The local parity block reduces the reconstruction overhead of data blocks because when a data block is lost, it can be reconstructed using only the blocks from its local group, rather than needing to access blocks across the entire storage system.

This greatly reduces the amount of data that needs to be read from the network during reconstruction, thereby reducing network traffic and speeding up the repair process. This is particularly beneficial in large-scale distributed storage systems where network bandwidth is often a limiting factor.

61. Evaluate and compare the performance of an LRC(12, 2, 2) and an RS(10, 4) in terms of storage overhead, reconstruction overhead, and fault tolerance.

In the context of distributed storage systems, both Local Reconstruction Codes (LRCs) and Reed-Solomon (RS) codes are forms of erasure coding used to ensure data reliability and durability. They help to recover lost data when certain parts of the data or nodes in the system fail.

LRC and RS are denoted as LRC(n, k, l) and RS(n, k) respectively. Here:

- n represents the total number of blocks after encoding: data blocks plus parity blocks.
- k is the number of data blocks.
- l (only for LRC) is the number of local parity blocks.
- The number of parity blocks in RS is n-k, and for LRC, it is n-k-l (global parity blocks) + l (local parity blocks).

For LRC(12, 2, 2), we have 2 data blocks, 2 local parity blocks, and 12-2-2 = 8 global parity blocks.

For RS(10, 4), we have 4 data blocks and 10-4 = 6 parity blocks.

1. **Storage Overhead**: Storage overhead is the extra storage required to store parity blocks in addition to the original data blocks.

   For LRC(12, 2, 2), the storage overhead is (12-2)/2 = 5. That means we need 5 times the original data size.

   For RS(10, 4), the storage overhead is (10-4)/4 = 1.5. That means we need 1.5 times the original data size.

   Therefore, RS(10, 4) has lower storage overhead.

2. **Reconstruction Overhead**: Reconstruction overhead is the amount of data that needs to be read from the network during data reconstruction.

   For LRC(12, 2, 2), when a data block is lost, it can be reconstructed using only the blocks from its local group, which are 2 blocks.

   For RS(10, 4), if a data block is lost, we need to read all remaining blocks, which are 9 blocks.

   Therefore, LRC(12, 2, 2) has lower reconstruction overhead.

3. **Fault Tolerance**: Fault tolerance is the ability of the system to continue functioning in the event of failures.

   For LRC(12, 2, 2), the system can tolerate up to 2 local failures without accessing global parities. For more failures, it depends on the number of global parity blocks, which is 8 here. So, it can handle up to 10 failures in total.

   For RS(10, 4), the system can tolerate up to 6 failures (equal to the number of parity blocks).

   Therefore, LRC(12, 2, 2) has higher fault tolerance.

To summarize, while RS(10, 4) offers less storage overhead, LRC(12, 2, 2) provides less reconstruction overhead and higher fault tolerance. The choice between the two would depend on the specific needs and constraints of your storage system.