Contents

Distributed Systems Exam Preparation

This repository is a collection of answers to commonly asked questions from past Distributed Systems exams. The answers were primarily generated using GPT-4o and refined using references from the course textbook to ensure accuracy and alignment with the material taught by the professor.

Structure of the Repository

- **Questions and Answers**: Each question from past exams has a corresponding answer provided in this repository. The answers aim to be comprehensive yet concise, making it easier for students to review and understand key concepts in Distributed Systems.
- **Reference Material**: The `res` folder contains an **unofficial course textbook**, specifically named **FINAL - DS notes col.pdf**, which serves as a reference for providing accurate and professor-aligned responses. This text

is crucial for ensuring that explanations and terminologies are consistent with what has been covered in the course.

  - [View FINAL - DS notes col.pdf](./res/FINAL%20-%20DS%20notes%20col.pdf)

How Answers Were Generated

1. **Initial Answers Using GPT-4o**: Most of the answers were generated using GPT-4o, known for its depth of understanding and ability to handle complex technical content.
2. **Refinement Using Textbook References**: For questions where the initial GPT-4o answers seemed imprecise or deviated from the course material, I provided the relevant sections from **FINAL - DS notes col.pdf** as a reference. These sections were used to refine and correct the answers, ensuring they are accurate and in line with the professor's teachings.

Compiling the Repository into a PDF

This repository contains Markdown files in various subdirectories (excluding `res`). You can compile these Markdown files, including LaTeX code, into a single PDF with a table of contents based on the directory and file names. The compiled PDF will include anchor links for easier navigation.

Steps to Compile:

1. **Install Required Tools**:
   - Install `pandoc`: `sudo apt install pandoc` (Linux) or use the package manager for your OS.
   - Install LaTeX distribution (e.g., `TeX Live` or `MikTeX`): `sudo apt install texlive-full`.
2. **Run the Compile Script**: Execute the following command in the root directory of the repository:

```
pandoc -s --toc --toc-depth=2 -o Distributed_Systems_Exam_Preparation.pdf \\
$(find exercises notes pastExams questions -name '*.md' | sort)
```

   - `--toc`: Generates a table of contents.
   - `--toc-depth=2`: Sets the depth of the table of contents to include up to two levels.
   - `-o`: Specifies the output PDF file name.
   - `find ...`: Locates all Markdown files in the specified directories.
3. **Open the PDF**: Once compiled, open `Distributed_Systems_Exam_Preparation.pdf` to review the document.

How to Use This Repository

1. **Review Questions and Answers**: Go through the questions and corresponding answers to solidify your understanding of key Distributed Systems concepts.
2. **Cross-Reference with Textbook**: Use the **FINAL - DS notes col.pdf** in the `res` folder to dive deeper into specific concepts or to verify the accuracy of an answer.
3. **Contribute**: If you find an answer that could be improved or notice any discrepancies, feel free to open an issue or submit a pull request with your suggestions.

Acknowledgments

- **Alireza**: For gathering the questions from past exams, which provided a foundation for this resource.
- **ChatGPT & GPT-4o**: For generating initial answers and assisting in the learning process.
- **Professor's Material**: The **unofficial course notes** in **FINAL - DS notes col.pdf** were instrumental in ensuring the accuracy and relevancy of the provided answers.

Disclaimer

This repository is a student-driven resource meant to aid in exam preparation. While efforts have been made to ensure the accuracy of the answers, always refer to the course material and the professor's notes for the most authoritative information. The **FINAL - DS notes col.pdf** is an **unofficial compilation** and should be used as a supplementary resource.

Happy studying and good luck on your exams!

```
# Table of Contents
- \[exercises/DHT\](#exercises-DHT)
- \[exercises/DistributedSnapshot\](#exercises-DistributedSnapshot)
- \[exercises/recovery_graphs_explanation\](#exercises-recovery_graphs_explanation)
- \[notes/Consistency_and_Replication/Consistency_Constraints_FIFO_Causal_Sequential\](#note
- \[questions/Big_Data/Dataflow_Programming_Model\](#questions/Big_Data-Dataflow_Programming
- \[questions/Big_Data/MapReduce_for_Product_Ratings\](#questions/Big_Data-MapReduce_for_Pro
- \[questions/Big_Data/dataflow_model_architectures\](#questions/Big_Data-dataflow_model_arc
- \[questions/Big_Data/stream_processing_comparison\](#questions/Big_Data-stream_processing_
- \[questions/Communication/Communication_Middleware_Characteristics\](#questions/Communicat
- \[questions/Communication/Message_Queuing_vs_RPC_Implementation\](#questions/Communicatio
- \[questions/Communication/Message_Queuing_vs_RPC_and_Client_Server_Interaction\](#questio
- \[questions/Consistency_and_Replication/Data_Reliability_with_Crash_and_Byzantine_Failures
- \[questions/Consistency_and_Replication/raft_consensus_protocol\](#questions/Consistency_a
```

- \[questions/Consistency_and_Replication/raft_vs_two_phase_commit\](#questions/Consistency_
- \[questions/Consistency_and_Replication/single_vs_raft_data_store\](#questions/Consistency
- \[questions/Consistency_and_Replication/two_phase_commit_safety_liveness\](#questions/Cons
- \[questions/Fault_Tolerance/Consensus_with_Process_Failures\](#questions/Fault_Tolerance-C
- \[questions/Fault_Tolerance/FloodSet_Algorithm_for_Consensus\](#questions/Fault_Tolerance-
- \[questions/Fault_Tolerance/Reliable_Group_Communication_with_Process_Failures\](#questio
- \[questions/Fault_Tolerance/Types_of_Failures_in_Distributed_Systems\](#questions/Fault_To
- \[questions/Modeling/Mobile_Code_Paradigms\](#questions/Modeling-Mobile_Code_Paradigms)
- \[questions/Naming/Approaches_to_Flat_Naming_LAN_WAN\](#questions/Naming-Approaches_to_Fla
- \[questions/Naming/Approaches_to_Remove_Unreferenced_Entities\](#questions/Naming-Approach
- \[questions/Naming/Flat_vs_Structured_Naming_Hierarchy\](#questions/Naming-Flat_vs_Structu
- \[questions/Naming/Structured_Naming_with_Server_Hierarchy\](#questions/Naming-Structured_
- \[questions/Peer_to_Peer/Chord_Lookup_Protocol\](#questions/Peer_to_Peer-Chord_Lookup_Prot
- \[questions/Peer_to_Peer/Parameter_Passing_RPC_RMI\](#questions/Peer_to_Peer-Parameter_Pas
- \[questions/Peer_to_Peer/Pub_Sub_vs_RPC_and_DHT_Dispatching\](#questions/Peer_to_Peer-Pub_
- \[questions/Peer_to_Peer/Query_Flooding_vs_Structured_P2P_Search\](#questions/Peer_to_Peer
- \[questions/Peer_to_Peer/p2p_query_flooding_vs_chord\](#questions/Peer_to_Peer-p2p_query_f
- \[questions/Streaming_Service_IP_Protocol_Limitations\](#questions-Streaming_Service_IP_Pr
- \[questions/Synchronization/Causal_vs_Total_Ordering\](#questions/Synchronization-Causal_t
- \[questions/Synchronization/Clock_Synchronization_Approaches\](#questions/Synchronization
- \[questions/Synchronization/Pessimistic_Timestamp_Ordering_Protocol\](#questions/Synchroni
- \[questions/Synchronization/Scalar_Clocks_Totally_Ordered_Multicast\](#questions/Synchroni
- \[questions/Synchronization/Scalar_Clocks_for_Causal_Ordering\](#questions/Synchronizatio
- \[questions/Synchronization/Vector_Clocks_for_Causal_Ordering\](#questions/Synchronizatio
- \[questions/Synchronization/Virtual_Synchrony_Concept_and_Implementation\](#questions/Sync

```yaml

## \[exercises/DHT\](#exercises-DHT)
<a id='exercises-DHT'></a>


**Question**: three peers (IDs0= 0, 7, 12) participate in a circular DHT with finger table u
assume that the DHT uses 4-bits to represent the node IDs and keys. show the routing table c

**Answer**:
First, recall the basics of a 4-bit Chord ring:

- The Chord ring has 16 possible IDs (0 through 15).
- Each node \\( n \\) maintains up to \\(\\log_2(16) = 4\\) finger entries.
- The \\(k\\)-th finger of node \\(n\\) (where \\(k\\) runs from 1 to 4) points to the succe
  \\\[
    (n + 2^\{k-1\}) \\mod 16.
  \\\]

We have three live nodes in the ring:
```

```
\\\[
\\\{\\,0,\\,7,\\,12\\\}.
\\\]

```yaml

### 1. Finger Table for Node 0

We compute the 4 fingers as follows:

1. \\(0 + 2^\{0\} = 1 \\mod 16 = 1\\)
   - Successor of ID 1 is the first live node \\(\\geq 1\\) on the ring, which is node 7.
2. \\(0 + 2^\{1\} = 2 \\mod 16 = 2\\)
   - Successor of ID 2 is node 7.
3. \\(0 + 2^\{2\} = 4 \\mod 16 = 4\\)
   - Successor of ID 4 is node 7.
4. \\(0 + 2^\{3\} = 8 \\mod 16 = 8\\)
   - Successor of ID 8 is the first live node \\(\\geq 8\\), which is node 12.

Hence Node 0's finger table is:

| Finger k | Start \\((0 + 2^\{k-1\})\\mod 16\\) | Successor |
|----------|-----------------------------------|-----------|
| k = 1    | 1                                 | 7         |
| k = 2    | 2                                 | 7         |
| k = 3    | 4                                 | 7         |
| k = 4    | 8                                 | 12        |

```yaml

### 2. Finger Table for Node 7

We compute similarly for node 7:

1. \\(7 + 2^\{0\} = 8 \\mod 16 = 8\\)
   - Successor of ID 8 is node 12.
2. \\(7 + 2^\{1\} = 9 \\mod 16 = 9\\)
   - Successor of ID 9 is node 12.
3. \\(7 + 2^\{2\} = 11 \\mod 16 = 11\\)
   - Successor of ID 11 is node 12.
4. \\(7 + 2^\{3\} = 15 \\mod 16 = 15\\)
   - Successor of ID 15 is the first live node \\(\\geq 15\\). None is \\(\\geq 15\\), so we

Hence Node 7's finger table is:

| Finger k | Start \\((7 + 2^\{k-1\})\\mod 16\\) | Successor |
```

```
|----------|--------------------------------|-----------|
| k = 1    | 8                              | 12        |
| k = 2    | 9                              | 12        |
| k = 3    | 11                             | 12        |
| k = 4    | 15                             | 0         |
```

```yaml
```

3. Finger Table for Node 12

Finally, for node 12:

1. \\(12 + 2^\{0\} = 13 \\mod 16 = 13\\)
   - Successor of ID 13 is node 0 (the next live node after 12 is 0).
2. \\(12 + 2^\{1\} = 14 \\mod 16 = 14\\)
   - Successor of ID 14 is node 0.
3. \\(12 + 2^\{2\} = 16 \\mod 16 = 0\\)
   - Successor of ID 0 is node 0 itself.
4. \\(12 + 2^\{3\} = 20 \\mod 16 = 4\\)
   - Successor of ID 4 is node 7 (the first node \\(\\geq 4\\) is 7).

Hence Node 12's finger table is:

```
| Finger k | Start \\((12 + 2^\{k-1\})\\mod 16\\) | Successor |
|----------|--------------------------------|-----------|
| k = 1    | 13                             | 0         |
| k = 2    | 14                             | 0         |
| k = 3    | 0                              | 0         |
| k = 4    | 4                              | 7         |
```

```yaml
```

Summary of Finger Tables

Node 0
- Finger\[1\] = 7
- Finger\[2\] = 7
- Finger\[3\] = 7
- Finger\[4\] = 12

Node 7
- Finger\[1\] = 12
- Finger\[2\] = 12
- Finger\[3\] = 12
- Finger\[4\] = 0

Node 12
- Finger\[1\] = 0
- Finger\[2\] = 0
- Finger\[3\] = 0
- Finger\[4\] = 7

These tables reflect how each node will forward lookups for IDs in a 4-bit Chord ring contai

```yaml

**Question**: peer 12 wants to retrieve the value of an object having key 15. show the excha

**Answer**:
First, recall **who** is responsible for key 15 in this 3-node Chord ring (with IDs 0, 7, 12

- Node 0's predecessor is 12.
- Therefore, node 0 is responsible for keys in \\((12,0\]\\) modulo 16.
- The numeric interval \\((12,0\]\\) in mod 16 is \\(\\{13,14,15,0\\}\\).
- Hence **node 0** holds the object whose key is 15.

```yaml

### How Peer 12 Routes a Lookup for Key = 15

Below is a simplified description of the Chord "lookup(key)" procedure:

1. **Check if key is in \\((n, \\text\{successor\}(n)\]** (mod \\(2^m\\)).
   - If yes, forward the request directly to \\(\\text\{successor\}(n)\\).
   - If no, forward the request to the "best finger" that precedes the key.

Given:
- Node 12's successor is node 0.
- Key = 15 lies in \\((12, 0\]\\) on the circular identifier space.

Therefore, node 12 immediately sees that 15 is in \\((12,0\]\\) and forwards the request to

### Message Flow

1. **Node 12 → Node 0**: "Lookup(key=15)"
   - Node 12 sends a lookup request to node 0, because 0 is the successor of 12 and covers k

2. **Node 0 checks responsibility**
   - Node 0 sees it is responsible for key=15 (since 15 is in \\((12,0\]\\)).

3. **Node 0 → Node 12**: returns the value
   - Node 0 replies back to node 12 with the stored value for key=15.

```
Hence, only **two messages** are needed:
1. A "find the key" (lookup) request from node 12 to node 0
2. A "here is your value" reply from node 0 back to node 12


```yaml

## \[exercises/DistributedSnapshot\](#exercises-DistributedSnapshot)
<a id='exercises-DistributedSnapshot'></a>
## QA Session around 00:15:00 - Page 80 Final DS Notes


In distributed snapshot algorithms like the **Chandy-Lamport algorithm**, a node stops recor

1. **When it receives a marker on that channel**:
   When a node receives a marker on a particular incoming channel, it:
   - Records the state of that channel as everything received before the marker.
   - Stops recording messages from that channel afterward.

2. **If no other messages arrive on that channel before the marker**:
   If a node receives a marker on an incoming channel and there were no messages received or

This ensures that:
- Each channel's state is captured as the set of messages received before the marker.
- Once the marker is received, recording for that specific channel ceases.

This mechanism helps maintain a **consistent global state** across all nodes in the distribu
```yaml

## \[exercises/recovery_graphs_explanation\](#exercises-recovery_graphs_explanation)
<a id='exercises-recovery_graphs_explanation'></a>
### Question:

Calculate the recovery line for the two diagrams below using the rollback-dependency graph f

```yaml

### Answer:

#### Recovery Line Calculation:
1. **Rollback-Dependency Graph**:
   - From the dependency graph, transform dependencies between intervals into dependencies b
   - Mark nodes corresponding to failure states.
   - Follow arrows from the marked nodes to identify states reachable from them.
   - Remove these marked states and determine the last unmarked checkpoint for each process.
```
```

```
   - The recovery line is defined by these last unmarked checkpoints.

2. **Checkpoint-Dependency Graph**:
   - Start with dependencies between the initial state of the sender interval and the final
   - Identify if a dependency makes the cut inconsistent.
   - If a cut is inconsistent, remove the state receiving the dependency arrow.
   - Continue eliminating dependent states until all remaining checkpoints are independent.
   - The recovery line is defined by the last independent checkpoints for all processes.

```yaml

#### When These Graphs Are Used:
Both graphs are used in **failure recovery** for distributed systems to ensure consistency:
- **Rollback-Dependency Graph**: Suitable for systems with **event logging** to analyze whic
- **Checkpoint-Dependency Graph**: Useful when **periodic checkpointing** is implemented, en

```yaml

#### How Data is Collected:
- **Rollback-Dependency Graph**:
  - Data is collected from **event logs**, which include:
    - Messages exchanged between processes.
    - State transitions and recorded events.
  - Dependency relationships between intervals are derived from these logs.

- **Checkpoint-Dependency Graph**:
  - Data is collected from **checkpoint metadata**, which includes:
    - Timestamps or identifiers of intervals between checkpoints.
    - Information piggybacked with messages, noting interval dependencies.
  - These records are used to build the graph and identify consistent cuts.

Both approaches are crucial for **backward recovery** in distributed systems, ensuring the s
```yaml

## \[notes/Consistency_and_Replication/Consistency_Constraints_FIFO_Causal_Sequential\](#not
<a id='notes/Consistency_and_Replication-Consistency_Constraints_FIFO_Causal_Sequential'></a
To check if this set of read/write operations satisfies **FIFO**, **Causal**, and **Sequenti

### 1. FIFO (First-In-First-Out) Consistency
- **Definition**: Each process must observe operations from any other single process in the
- **Constraint**:
  - For any two operations `W(x)v1` and `W(x)v2` by the same process where `v1` occurs befor
  - In this example, each process should observe writes from other processes in the same orc

### 2. Causal Consistency
- **Definition**: If one operation causally depends on another (e.g., a read depends on a pr
```
```

```
- **Constraint**:
  - If `W(x)v1` happens before `R(x)v2` and `R(x)v2` reads the value from `W(x)v1`, then all
  - Similarly, if `W(x)v1` by process `P0` happens before `W(x)v2` by `P1`, then all process
  - In this example, any read that observes a particular write must respect that write's orc

### 3. Sequential Consistency
- **Definition**: There exists a global ordering of all operations such that each process's
- **Constraint**:
  - There must be a single global order of operations that is respected by all processes.
  - This means if `W(x)v1` occurs before `W(x)v2` in this global order, all processes must c
  - For this example, all processes would need to see the same total order of all read and w

```yaml

Using these constraints, you can analyze the sequence of operations for each process and det
```yaml

## \[questions/Big_Data/Dataflow_Programming_Model\](#questions/Big_Data-Dataflow_Programmi
<a id='questions/Big_Data-Dataflow_Programming_Model'></a>

**Question:** *Describe the dataflow programming model and explain why it is suitable for la

```yaml

### Answer

#### 1. **Dataflow Programming Model**
The **dataflow programming model** is a paradigm in which the program execution is driven by
- **Nodes** represent computational tasks or operations.
- **Edges** represent data dependencies between these tasks.

Each node executes as soon as all of its input data is available, making the dataflow model
- **Parallelism**: Since operations can execute independently and concurrently whenever data
- **Scalability**: Easily scalable across distributed systems as different tasks can be assi

```yaml

#### 2. **Suitability for Large-Scale Distributed Data Processing**
The dataflow programming model is well-suited for large-scale distributed data processing be
- **Parallel Execution**: Tasks can be executed in parallel across multiple machines, taking
- **Fault Tolerance**: Many dataflow systems are designed to handle failures gracefully, rec
- **Optimized Data Handling**: Data is efficiently transferred between tasks, reducing unnec
- **Flexible Resource Allocation**: Distributed environments can dynamically allocate resour

```yaml
```
```

```
#### 3. **Architectural Approaches to Implement Dataflow-Based Systems**

1. **Batch Processing Architecture**
   - **Overview**: In batch processing, data is collected over a period of time and processe
   - **How It Works**:
     - Data is divided into chunks (batches) and fed through the dataflow graph.
     - Intermediate results are typically stored in distributed storage systems like HDFS (H
     - The computation is fault-tolerant, as it can recompute lost data chunks from the inpu
   - **Adoption for Stream Processing**:
     - **Challenges**: Batch processing systems are not natively designed for real-time data
     - **Adaptation**: Frameworks like **Apache Spark Streaming** provide micro-batching, wh

2. **Stream Processing Architecture**
   - **Overview**: In stream processing, data is continuously processed as it arrives, with
   - **How It Works**:
     - Data is processed immediately as it becomes available, without waiting for a complete
     - The system maintains state information across events, enabling real-time analytics an
     - Checkpointing and distributed snapshots are often used to ensure fault tolerance.
   - **Adoption for Stream Processing**:
     - **Natively Suitable**: Stream processing architectures are explicitly designed for ha
     - **Efficient State Management**: They provide features like windowed operations and ev

```yaml

### Conclusion
The **dataflow programming model** is inherently parallel and scalable, making it ideal for
```yaml

## \[questions/Big_Data/MapReduce_for_Product_Ratings\](#questions/Big_Data-MapReduce_for_Pr
<a id='questions/Big_Data-MapReduce_for_Product_Ratings'></a>

**Question:** *Show, using pseudo-code, how to use MapReduce to compute the average rating o

```yaml

### Answer

To compute the average rating for each product using the MapReduce paradigm, we can break do

```yaml

### Pseudo-Code Implementation

```python
# Pseudo-code for MapReduce to compute the average rating for each product
```
```

```
# The input dataset consists of entries: (user, product, rate)

# Step 1: Map Function
def map_function(entry):
    # Split the entry into user, product, and rate
    user, product, rate = entry

    # Emit (product, (rate, 1))
    emit(product, (rate, 1))

# Step 2: Reduce Function
def reduce_function(product, values):
    # Initialize variables for sum of ratings and count of ratings
    total_rate = 0
    total_count = 0

    # Iterate through all values received for this product
    for rate, count in values:
        total_rate += rate
        total_count += count

    # Compute the average rating
    average_rate = total_rate / total_count

    # Emit the product and its average rating
    emit(product, average_rate)

# The system will automatically handle the parallelization:
# - The Map phase will distribute the entries to different nodes, which will execute map_fu
# - The intermediate results (product, (rate, 1)) are shuffled and sent to the appropriate
# - The Reduce phase aggregates the ratings and computes the average
```

“‘yaml

Explanation of the Algorithm

1. **Map Phase**:
   - The `map_function` takes each entry from the dataset, which consists of a user ID, product ID, and the rating given by the user.
   - It extracts the `product` and `rate` from each entry and emits a key-value pair:
     - **Key**: The `product` ID
     - **Value**: A tuple containing the `rate` and the count `1` (to keep track of how many ratings are being aggregated)
   - This phase distributes the data across multiple nodes in the cluster for parallel processing.

2. **Shuffle and Sort**:
    - The MapReduce framework automatically handles shuffling and sorting the output of the Map phase.
    - All key-value pairs with the same `product` ID are grouped together and sent to the same reducer.
3. **Reduce Phase**:
    - The `reduce_function` receives each product and a list of all `(rate, 1)` tuples for that product.
    - It aggregates the total ratings (`total_rate`) and the total count of ratings (`total_count`).
    - The average rating for the product is calculated as `total_rate / total_count`.
    - The final output is the `product` ID and its `average_rate`.

“‘yaml

Assumptions and Scalability Considerations

1. **Large Dataset**: The dataset is assumed to be large, so it is distributed across multiple nodes for efficient processing.
2. **Parallel Execution**: The Map and Reduce phases can be executed in parallel on different nodes, making this approach scalable.
3. **Distributed Infrastructure**: The algorithm is designed for a distributed system, where data is processed in chunks across a cluster of machines.

“‘yaml

Scenario Discussion

The MapReduce approach efficiently handles large datasets by dividing the computation into independent tasks that can be processed concurrently. This is especially useful for analyzing product ratings on a large e-commerce platform where the data may span multiple servers or storage units. “‘yaml

[questions/Big_Data/dataflow_model_architectures](#questions/Big_Data-dataflow_model_architectures)

Question:
Consider the Dataflow model for big data processing. Describe the key characteristics of the model. Describe what are the two architectures to implement it: scheduling of tasks and pipelining of tasks.

“‘yaml

Answer

1. Key Characteristics of the Dataflow Model The **Dataflow model** is an approach to Big Data processing where computation is represented as a

directed acyclic graph (DAG) of operations. The nodes in the graph represent transformations, and the edges represent the flow of data between them.

- **Characteristics**:
  1. **DAG Representation**:
     - Computations are expressed as a graph, where data flows through a series of transformations.
     - Examples of transformations: map, filter, join, groupBy.
  2. **Decoupling of Computation and Data**:
     - Data and operations are independent, promoting modularity.
  3. **Scalability**:
     - The model is inherently scalable, allowing computations to be distributed across multiple machines.
  4. **Support for Fault Tolerance**:
     - By maintaining lineage or checkpoints, the model ensures that failed operations can be recomputed or restarted.
  5. **Optimized for Parallelism**:
     - Operations can run in parallel wherever dependencies allow, increasing throughput.
  6. **Applicability**:
     - Used for both batch processing (processing all data at once) and stream processing (processing data continuously as it arrives).

“‘yaml

2. Two Architectures to Implement the Dataflow Model

1. **Scheduling of Tasks (Batch or Micro-Batch Processing)**:
   - This architecture divides the computation graph into **stages**. Each stage processes a batch of data and produces intermediate results that are passed to the next stage.
   - **How It Works**:
     - Tasks are scheduled dynamically, ensuring that they run close to the data to minimize data movement.
     - Intermediate results may be stored in memory or on disk, depending on the system.
   - **Example**:
     - **Apache Spark** uses this approach with its micro-batch processing model.
   - **Advantages**:
     - Optimizations can be applied at the scheduling level, such as data compression and load balancing.
     - Better resource management as tasks can be rescheduled dynamically.
   - **Drawbacks**:
     - Higher latency, especially for real-time data, as tasks need to wait for batching to complete.

2. **Pipelining of Tasks (Continuous Processing)**:
   - This architecture processes data **continuously** as it flows through the DAG. Operators in the graph are instantiated and begin processing as soon as data becomes available.
   - **How It Works**:
     - Each operator communicates directly with downstream operators via TCP channels or similar mechanisms, without intermediate storage.
     - Results are produced incrementally, reducing overall latency.
   - **Example**:
     - **Apache Flink** uses this approach for streaming data.
   - **Advantages**:
     - Lower latency as data does not wait for batch formation.
     - Suitable for real-time analytics and event-driven applications.
   - **Drawbacks**:
     - Load balancing and elasticity are more challenging because tasks are statically assigned at deployment time.

“‘yaml

3. Comparison: Scheduling vs. Pipelining

Aspect	Scheduling of Tasks	Pipelining of Tasks
Latency	Higher, due to batching and task scheduling overhead.	Lower, as data flows continuously through the pipeline.
Throughput	High throughput due to task optimization and batching.	Lower throughput due to smaller, continuous data chunks.
Fault Tolerance	Relies on lineage or checkpoints to recompute tasks.	Relies on frequent checkpoints to resume from failures.
Load Balancing	Dynamic scheduling enables better load balancing.	Static assignment limits load balancing flexibility.
Elasticity	Easier to adjust resources dynamically.	Difficult, as tasks are pre-assigned. Requires system restarts.

“‘yaml

Conclusion

The **Dataflow model** provides a powerful abstraction for distributed big data processing, enabling both batch and stream processing. The choice between **scheduling of tasks** and **pipelining of tasks** depends on the application

requirements: - Scheduling is better for high throughput and dynamic resource management. - Pipelining excels in low-latency scenarios, such as real-time analytics.

“‘yaml

[questions/Big_Data/stream_processing_comparison](#questions/Big_Data-stream_processing_comparison)

Question:
a. Describe what is stream processing,
b. Discuss two alternative approaches to implement it, and
c. Compare the two approaches in terms of latency and load balancing.

“‘yaml

Answer

a. What is Stream Processing?

- **Definition**:
  - Stream processing is a method of handling continuous flows of data in real-time or near real-time. Instead of processing data in large static batches, stream processing operates on data as it is generated or received.
- **Key Characteristics**:
  - **Low Latency**: Designed to process data with minimal delay.
  - **Continuous Processing**: Handles data in a continuous, unbounded manner.
  - **Applications**: Real-time analytics, event detection, monitoring systems, and more.

“‘yaml

b. Two Approaches to Stream Processing

1. **Micro-Batch Processing (Apache Spark)**:
   - The input data stream is divided into small batches, and each batch is processed independently.
   - State is maintained across batches to handle aggregation or updates.
   - Example: Streaming word count where the count of each word is persisted and updated batch-by-batch.
2. **Continuous or Pipelined Processing (Apache Flink)**:
   - Operators are instantiated as soon as the job is submitted, creating a topology of tasks.
   - Each operator processes data as it becomes available and communicates with others via TCP channels.

- Data flows continuously through the network without waiting for batching or scheduling.

“‘yaml

c. Comparison: Micro-Batch vs. Continuous Processing

Aspect	Micro-Batch Processing (Spark)	Continuous Processing (Flink)
Latency	Higher latency due to batching and scheduling overhead.	Lower latency as data flows directly between operators without waiting for batches.
Load Balancing	Dynamic scheduling allows better load balancing. Tasks can be reallocated based on data distribution.	Static allocation of tasks at job deployment makes load balancing harder.
Throughput	Higher throughput due to opportunities for optimizing large data transfers and compression.	Lower throughput as data is processed in smaller chunks without such optimizations.
Elasticity	Supports dynamic resource scaling. Tasks can be moved and rescheduled as needed.	Static topology limits elasticity. Requires snapshots and restarts for resource changes.
Fault-Tolerance	Uses lineage to recompute lost data. No need for replication.	Relies on periodic checkpointing. Restarts from the last checkpoint if failures occur.

“‘yaml

Conclusion

- Stream processing can be implemented via **micro-batch processing (Spark)** or **continuous processing (Flink)**, each with distinct trade-offs.
- **Micro-Batch Processing** is suitable for high-throughput applications that can tolerate some latency and require better elasticity and load balancing.
- **Continuous Processing** excels in scenarios where low latency is critical, such as real-time analytics or monitoring systems.

“‘yaml

[questions/Communication/Communication_Middleware_Characteristics](#questi Communication_Middleware_Characteristics)

Question: *Consider the characteristics of a communication middleware: explicit vs. implicit addressing, sync vs. async, transient vs. persistence, unicast vs. multicast. Associate those characteristics with the three architectural styles: REST, Pub/Sub, and Message Queuing. Briefly explain.*

“‘yaml

Answer

1. Characteristics of Communication Middleware

1. **Explicit vs. Implicit Addressing**:
   - **Explicit Addressing**: The sender specifies the exact recipient(s) for the message.
   - **Implicit Addressing**: The sender does not specify the exact recipient(s); instead, messages are routed based on topics or content.
2. **Synchronous vs. Asynchronous**:
   - **Synchronous**: The sender waits for a response from the recipient before continuing.
   - **Asynchronous**: The sender does not wait for a response and continues execution immediately after sending the message.
3. **Transient vs. Persistent**:
   - **Transient**: Messages are only stored temporarily and may be lost if the recipient is unavailable.
   - **Persistent**: Messages are stored until they are successfully delivered, ensuring reliability.
4. **Unicast vs. Multicast**:
   - **Unicast**: Messages are sent to a single recipient.
   - **Multicast**: Messages are sent to multiple recipients simultaneously.

“‘yaml

2. Associating Characteristics with Architectural Styles

1. **REST (Representational State Transfer)**

   - **Addressing**: **Explicit**. The client explicitly addresses requests to specific resources using URLs.
   - **Synchronization**: **Synchronous**. Typically, REST calls are synchronous, where the client waits for a response from the server.
   - **Message Duration**: **Transient**. The request and response are ephemeral, and no persistence is maintained beyond the transaction.
   - **Communication**: **Unicast**. REST communication is generally one-to-one, from a client to a server.

Brief Explanation: REST provides a simple, stateless way to access resources with clear, explicit addressing through HTTP URLs. It is mostly synchronous and unicast, suitable for direct client-server interactions.

2. **Pub/Sub (Publish/Subscribe)**

   - **Addressing**: **Implicit**. Publishers do not specify recipients; instead, they publish messages to a topic or event channel, and subscribers receive messages based on their subscriptions.
   - **Synchronization**: **Asynchronous**. Messages are published and delivered asynchronously, allowing for decoupled communication.
   - **Message Duration**: **Transient**. Messages are usually not stored if there are no active subscribers unless the system provides specific buffering.
   - **Communication**: **Multicast**. Messages can be delivered to multiple subscribers who have expressed interest in the topic.

   Brief Explanation: Pub/Sub is ideal for event-driven systems where components need to be loosely coupled. Publishers and subscribers do not need to know each other, and communication is multicast, enabling scalable message dissemination.

3. **Message Queuing**

   - **Addressing**: **Explicit**. Messages are sent to specific queues that are managed and monitored.
   - **Synchronization**: **Asynchronous**. The sender does not wait for the message to be processed, allowing for decoupled and efficient communication.
   - **Message Duration**: **Persistent**. Messages are stored in queues until they are processed, providing reliability and fault tolerance.
   - **Communication**: **Unicast** (though load balancing across multiple servers is possible).

   Brief Explanation: Message queuing provides reliable, persistent messaging with explicit addressing. It is well-suited for systems requiring guaranteed message delivery and the ability to handle varying processing loads.

“‘yaml

Conclusion

Each architectural style has unique characteristics suited to different communication needs. REST is straightforward and synchronous, Pub/Sub supports scalable multicast communication with implicit addressing, and message queuing ensures reliability with persistent and asynchronous messaging.

“‘yaml

[questions/Communication/Message_Queuing_vs_RPC_Implementation](#quest Message_Queuing_vs_RPC_Implementation)

Question: *Describe message queuing in general, the three differences with respect to RPC. Then explain how message queuing can be used to implement a client-server interaction and the advantages of this approach with respect to a traditional implementation based on RPC.*

“‘yaml

Answer

1. General Concept of Message Queuing

- **Message Queuing** is a communication model where messages are sent to a queue and are asynchronously processed by the recipient. Unlike direct message passing or synchronous calls, message queuing provides a layer of abstraction between the sender and the receiver.
- **Key Characteristics**:
  - **Asynchronous Communication**: The sender does not wait for the receiver to process the message, enabling the sender to continue its operations.
  - **Persistent Storage**: Messages are stored in the queue until the recipient retrieves and processes them, ensuring delivery even if the receiver is temporarily unavailable.
  - **Decoupling**: The sender and receiver are independent, allowing them to operate at different times and speeds.

“‘yaml

2. Three Key Differences Between Message Queuing and RPC

1. **Synchronous vs. Asynchronous**:
   - **RPC (Remote Procedure Call)**: Synchronous communication where the client sends a request and waits for the server's response before continuing execution.
   - **Message Queuing**: Asynchronous communication where the client sends a message to a queue and continues execution without waiting for an immediate response from the server.
2. **Tight Coupling vs. Loose Coupling**:
   - **RPC**: The client and server are tightly coupled, meaning they must be aware of each other's location and remain connected during the entire interaction.
   - **Message Queuing**: The client and server are loosely coupled, allowing for greater flexibility and independence, as they do not need to be simultaneously connected.
3. **Reliability and Fault Tolerance**:

- **RPC**: Less reliable in scenarios where the server is unavailable, as the client must handle failures directly.
- **Message Queuing**: Offers built-in reliability, as messages are persisted in the queue until they are successfully delivered and processed, making it more resilient to network or server failures.

“‘yaml

3. Implementing Client-Server Interaction with Message Queuing

- **Mechanism**:
  - The **client** sends requests by placing messages into the server's message queue.
  - The **server** retrieves messages from the queue asynchronously, processes them, and may send a response back to the client's designated response queue.
- **Example**:
  - In an e-commerce application, a client sends a message to process an order. The message is queued, and the server processes it when ready, updating the status asynchronously.

“‘yaml

4. Advantages of Message Queuing Over RPC

1. **Decoupling and Flexibility**:
   - The client and server do not need to be online or connected at the same time. This decoupling makes the system more flexible and easier to manage, especially in distributed environments.
2. **Improved Reliability**:
   - Since messages are stored in a queue, there is a guarantee of delivery even if the server experiences temporary downtime. This makes the system fault-tolerant and ensures that no messages are lost.
3. **Scalability and Load Balancing**:
   - Message queues can distribute requests among multiple servers, facilitating load balancing and enabling horizontal scaling. As the demand increases, more servers can be added to handle the queued messages efficiently.

“‘yaml

Conclusion

Message queuing provides a robust alternative to traditional RPC by offering asynchronous, loosely coupled communication that enhances system reliability and scalability. It is well-suited for distributed systems where components operate at different speeds or require resilience to network and server failures.

“‘yaml

[questions/Communication/Message_Queuing_vs_RPC_and_Client_Server_Int
Message_Queuing_vs_RPC_and_Client_Server_Interaction)

Question: Describe message queuing in general and the difference with respect to RPC. How message queuing can be used to implement a client-server interaction, and what are the advantages of this approach compared to traditional implementations with RPC.

Answer:

General Concept of Message Queuing:

Message Queuing is a communication model where messages are sent to a queue rather than directly to a recipient. It is a form of persistent, asynchronous communication where the sender and receiver are decoupled in both time and space. The message remains in the queue until the recipient retrieves and processes it.

Features: - **Asynchronous:** The sender does not need to wait for the recipient to be ready. Messages are queued and processed when the recipient is available. - **Persistent:** Messages are stored until delivery, ensuring reliability even if the recipient is temporarily unavailable. - **Decoupled Communication:** Sender and receiver do not need to be connected simultaneously.

“‘yaml

Difference Between Message Queuing and RPC:

RPC (Remote Procedure Call): - RPC is a synchronous communication model where a client makes a direct call to a remote procedure and waits for a response. It closely resembles local function calls but operates across network boundaries. - **Tightly Coupled:** The client and server must be aware of each other's location and remain connected during the call.

Message Queuing: - In contrast, message queuing decouples the client and server, allowing for asynchronous communication. The client sends a message to the queue and can continue executing without waiting for the server to respond. - **Loosely Coupled:** The sender and receiver are independent, providing greater flexibility and resilience.

“‘yaml

Implementing Client-Server Interaction with Message Queuing:

Mechanism: - The client sends requests to a message queue managed by the server. The server asynchronously fetches these requests, processes them, and sends the results back to the client's response queue. - **Example:** In a distributed application, a client may request a data retrieval operation by placing

a message in the server's queue. The server processes this request when it is ready and places the response in the client's designated response queue.

“‘yaml

Advantages of Message Queuing Over RPC:

1. **Decoupling:**
   - The client and server do not need to be simultaneously active. This is especially useful in scenarios where the client's availability does not match the server's.
2. **Reliability:**
   - Messages persist in the queue until they are processed, providing fault tolerance and ensuring that messages are not lost if a component crashes or goes offline temporarily.
3. **Load Balancing:**
   - Queues can be shared among multiple servers to distribute the processing load evenly. This enhances scalability and efficiency, especially under heavy loads.
4. **Flexibility and Scalability:**
   - Message queuing systems can easily scale by adding more servers to handle queued requests, without the need for changes to the client.

“‘yaml

Architectural Issues and Considerations:

- **Symbolic Names and Lookup Services:** Queues are identified by symbolic names, and a lookup service may be required to map these to network addresses.
- **Queue Managers and Relays:** Queue managers can act locally or remotely to manage message flow, often using application-level routing. This setup improves fault tolerance and can integrate with subsystems through brokers for message conversion.

“‘yaml

Conclusion:

Message queuing provides a more resilient and scalable architecture compared to traditional RPC. It decouples the client and server, offers reliable message delivery, and simplifies load balancing. However, it introduces complexity at the application layer, requiring careful management of queues and potential message routing strategies.

“‘yaml

[questions/Consistency_and_Replication/Data_Reliability_with_Crash_and_By Data_Reliability_with_Crash_and_Byzantine_Failures)

Question: *In a distributed system, a group of servers hold the same copy of an immutable data store. Clients contact one or more elements of the group to receive pieces of such data. How big must the group be to tolerate \(k \) crashes of servers? Describe the behavior to receive such data (clarify your assumptions). How would your answer change if the servers exhibited Byzantine behavior?*

“‘yaml

Answer

1. Group Size to Tolerate \(k \) Crashes To ensure data availability despite **crash failures** (fail-stop behavior where servers stop functioning but do not behave maliciously), the group of servers must be large enough to guarantee that data remains accessible even when \(k \) servers have failed.

- **Minimum Group Size**: To tolerate \(k \) crash failures, you need a total of: \[N \geq k + 1 \] This is because, even if \(k \) servers crash, there should still be at least one functioning server to serve the clients.

“‘yaml

2. Behavior to Receive Data

Assuming reliable communication and the following conditions:

1. **Data Redundancy**: All servers in the group hold the same, immutable copy of the data. There is no need for consistency mechanisms since the data does not change.
2. **Client Query Mechanism**:
   - **Contact Strategy**: Clients can query one or more servers to request pieces of the data.
   - **Redundancy Handling**: If a client contacts a server that has crashed, the client should be able to retry with another server in the group until it successfully retrieves the data.
   - **Assumptions**: We assume that clients are aware of all the servers in the group and that network communication is reliable.
3. **Data Retrieval**:
   - Clients may use a **simple retry mechanism** to handle server crashes:
     – The client sends a data request to a randomly selected server.
     – If the server does not respond within a predefined timeout, the client retries with another server until the data is successfully received or all available servers are contacted.

“‘yaml

3. Handling Byzantine Failures

If the servers exhibit **Byzantine behavior** (where servers may act maliciously, send corrupted data, or exhibit unpredictable behavior), the requirements to maintain data reliability become stricter.

Group Size to Tolerate \(k \) Byzantine Failures To tolerate \(k \) Byzantine failures, the group size \(N \) must satisfy: \[N \geq 3k + 1 \] This requirement ensures that even if \(k \) servers behave maliciously or send incorrect data, there will be enough honest servers to reliably reconstruct the correct data. This threshold comes from the need to use **Byzantine Fault Tolerance (BFT)** algorithms, which require a majority of non-faulty servers to reach a consensus on the correct data.

“‘yaml

4. Behavior to Receive Data Under Byzantine Conditions

1. **Data Verification**: Clients must implement mechanisms to verify the correctness of the data received from the servers. This could involve:
   - **Quorum-Based Reads**: The client queries multiple servers and compares the responses. If the majority of responses are consistent, the client can safely accept the data. The minimum size of the quorum must ensure that a majority of responses come from honest servers.
   - **Digital Signatures or Hashes**: Servers may provide cryptographic proofs (e.g., digital signatures or data hashes) that clients can use to verify data integrity.
2. **Redundant Queries**:
   - Clients must query at least \(2k + 1 \) servers to ensure that a majority of the responses (at least \(k + 1 \)) come from non-faulty servers, allowing the client to disregard the incorrect or malicious data from the \(k \) Byzantine servers.

“‘yaml

5. Summary of Changes Between Crash and Byzantine Failures

1. **Crash Failures**:
   - Minimum Group Size: \(N \geq k + 1 \)
   - Simple retry mechanism is sufficient to handle failures.
   - Data retrieval is straightforward because the only failure mode is a server not responding.
2. **Byzantine Failures**:
   - Minimum Group Size: \(N \geq 3k + 1 \)
   - Requires additional mechanisms for data verification, such as quorum-based reads or cryptographic verification.
   - The client must be able to tolerate and identify malicious or incorrect data and rely on the majority of honest servers.

“‘yaml

Conclusion

To tolerate \(k \) crash failures, the group size must be at least \(k + 1 \). For Byzantine failures, the group size must be significantly larger, at \(3k + 1 \), to ensure data integrity and reliability. The strategies for handling data retrieval also become more complex under Byzantine conditions, requiring mechanisms to verify the correctness of the data received from multiple servers. “‘yaml

[questions/Consistency_and_Replication/raft_consensus_protocol](#questions/C raft_consensus_protocol)

Question: *Consider the Raft Consensus Protocol. Which problem does it solve? Under which assumptions? Does the protocol guarantee safety (is it always correct) and liveness (it may always make progress)? Motivate your answers.*

“‘yaml

Answer

1. **Problem Solved by Raft Consensus Protocol**

   - The **Raft Consensus Protocol** is designed to solve the **distributed consensus problem**, where multiple nodes in a distributed system must agree on a single, consistent value (e.g., the order of operations in a replicated state machine).
   - Raft ensures:
     - **Consistency**: All non-faulty nodes agree on the same value or sequence of operations.
     - **Fault Tolerance**: The protocol can tolerate failures of up to half of the nodes in the system (majority quorum is required).

2. **Assumptions of the Protocol**

   - **Reliable Message Delivery**: Messages may be delayed or reordered but will eventually be delivered.
   - **Crash Recovery**: Nodes may fail and later recover with their stable storage intact.
   - **Majority Availability**: The system assumes that a majority of the nodes are operational at any given time.
   - **Synchronous Intervals**: While Raft operates in asynchronous systems, it assumes periodic intervals of synchrony where messages are delivered within predictable time bounds.

3. **Guarantees of Raft Protocol**

   - **Safety (Correctness)**:

  - Raft guarantees **safety** under all conditions, including node crashes or message delays.
  - Key mechanisms ensuring safety:
    * **Leader Election**: Only one leader is allowed at a time, avoiding conflicts.
    * **Log Matching Property**: All committed entries are consistent across all nodes.
    * **Commit Rules**: Entries are committed only when a majority acknowledges them.
    * Raft's design avoids split-brain scenarios by ensuring that only the leader with the most up-to-date log can successfully append entries.
- **Liveness (Progress)**:
  - **Liveness is not guaranteed** during certain failure scenarios.
  - Example:
    * In cases of **network partitions**, Raft may fail to elect a leader if a majority quorum cannot be established. Without a leader, the protocol cannot make progress until the partition is resolved.
    * If timers are misconfigured or there are frequent leader crashes, the system may experience delays in electing a new leader.

"'yaml

4. Scenario Demonstrating Lack of Liveness

- **Network Partition**:
  - Assume a cluster of 5 nodes. If a network partition isolates 3 nodes from 2 nodes:
    * The group of 3 nodes can still elect a leader and make progress (if they have a majority).
    * The group of 2 nodes will be unable to elect a leader, leading to a halt in operations for that subset.
    * If the partition persists, Raft may fail to achieve liveness for the entire system, as the minority group remains blocked.
- **Leader Crash with Network Instability**:
  - If a leader crashes and network instability causes delays in reaching a quorum for leader election, the system remains blocked until a new leader is successfully elected.

"'yaml

5. Conclusion

- Raft addresses the distributed consensus problem by providing **safety guarantees** under all conditions.
- **Liveness** is not always guaranteed, as the protocol may block during network partitions or if a majority of nodes are unavailable.

- These trade-offs are inherent in distributed systems due to the **CAP theorem**, where Raft prioritizes **Consistency** over **Availability** during failures.

“‘yaml

[questions/Consistency_and_Replication/raft_vs_two_phase_commit](#questio raft_vs_two_phase_commit)

The **Raft Consensus Protocol** and the **Two-Phase Commit (2PC) Protocol** are not the same thing. While both deal with achieving agreement in distributed systems, they address different problems and operate differently. Here's a comparison:

“‘yaml

1. **Purpose**

- **Raft Consensus Protocol**:
  - Solves the **distributed consensus problem**, ensuring that all nodes in a distributed system agree on a single, consistent state or sequence of operations (e.g., in a replicated state machine).
  - Typically used for **leader election** and maintaining a consistent log across distributed nodes (e.g., in distributed databases or configuration systems).
- **Two-Phase Commit (2PC)**:
  - Solves the problem of **atomicity in distributed transactions** by ensuring that either all nodes commit a transaction or none of them do.
  - It ensures that a distributed transaction is applied consistently across all participants.

“‘yaml

2. **Mechanism**

- **Raft Consensus Protocol**:
  - Operates as a state machine replication protocol with a **leader-based architecture**.
  - Uses three phases:
    1. **Leader Election**: A leader is elected to coordinate log replication.
    2. **Log Replication**: The leader receives client requests and appends them to its log, then replicates these entries to followers.
    3. **Commit**: Once a majority of followers acknowledge an entry, it is committed and applied to the state machine.
- **Two-Phase Commit (2PC)**:
  - A transaction protocol with a **coordinator-based architecture**.

- Involves two phases:
  1. **Prepare Phase**: The coordinator asks all participants if they can commit. Participants vote "yes" (if ready to commit) or "no."
  2. **Commit Phase**: If all participants vote "yes," the coordinator sends a "commit" message. If any participant votes "no," it sends an "abort" message.

"'yaml

3. Fault Tolerance

- **Raft**:
  - Designed to tolerate **node failures** and maintain progress as long as a majority of nodes are operational.
  - A leader crash triggers a **leader re-election**, allowing the system to continue making progress once a new leader is chosen.
- **2PC**:
  - **Not fault-tolerant**:
    - If the coordinator crashes during the commit phase, participants can be left in a blocked state, unsure whether to commit or abort.
    - This lack of liveness is a major limitation of 2PC.

"'yaml

4. Guarantees

- **Raft**:
  - Guarantees **consistency (safety)**, ensuring that all nodes agree on the same log.
  - Liveness depends on resolving network partitions and leader availability.
- **2PC**:
  - Guarantees **atomicity (all-or-nothing behavior)** for transactions.
  - Liveness is not guaranteed because it can block if the coordinator fails.

"'yaml

5. Use Cases

- **Raft**:
  - Consensus for replicated logs (e.g., databases like etcd or Consul).
  - Distributed state machine replication.
- **2PC**:
  - Distributed transactions in databases, ensuring all nodes apply the same transaction atomically.

"'yaml

Conclusion

While both Raft and 2PC ensure consistency, they operate at different levels and are used for different purposes. Raft is a consensus protocol for achieving agreement on a replicated log, while 2PC is a transactional protocol for achieving atomicity in distributed systems.

“‘yaml

[questions/Consistency_and_Replication/single_vs_raft_data_store](#questions/Consistency_and_Replication/single_vs_raft_data_store)

Question:
Consider a simple data store with two implementations:
a. The system is implemented using a single machine.
b. The system is replicated across 5 machines using the Raft Consensus Protocol to provide consistency across replicas.
Compare the two implementations in terms of response time for client requests, consistency, and fault tolerance.

“‘yaml

Answer

a. Single-Machine Implementation

- The data store is hosted entirely on a single machine.
- All client requests are handled directly by the single machine.

“‘yaml

b. Replicated System with Raft Consensus

- The data store is replicated across 5 machines, ensuring consistency via the Raft Consensus Protocol.
- One node acts as the leader, and all client requests are processed through this leader.

“‘yaml

Comparison

Aspect	Single-Machine Implementation	Replicated System (Raft)
Response Time	- Faster response time for client requests. - No need for network communication; all operations are local.	- Slower response time due to additional overhead from Raft. - Writes must propagate to a majority of nodes before committing.

Aspect	Single-Machine Implementation	Replicated System (Raft)
	- Read operations are instantaneous (limited to local I/O).	- Reads may involve querying the leader or other replicas (depending on configuration).
Consistency	- Consistent by default as there is only one copy of the data.	- Strong consistency guaranteed by Raft.
	- No risk of conflicts or divergence between replicas.	- Ensures a consistent view of data across replicas, even in the presence of failures.
	- No built-in mechanisms to handle concurrent updates (requires client logic).	- Raft resolves conflicts and enforces a single leader for coordination.
Fault Tolerance	- No fault tolerance: system becomes unavailable if the single machine fails.	- High fault tolerance: can tolerate failures of up to 2 nodes (majority quorum still available).
	- Data loss is likely if the machine crashes unless backups are taken frequently.	- Data is replicated across nodes, reducing the risk of data loss.

“‘yaml

Explanation of Differences

1. **Response Time**:
   - The single-machine implementation is inherently faster for client requests as there is no network communication or coordination overhead.
   - In the replicated system, Raft requires communication between the leader and a majority of replicas for write operations, increasing latency.
2. **Consistency**:
   - The single-machine implementation provides consistency automatically as there's only one data source.
   - The Raft-based system guarantees strong consistency but with the added complexity of consensus protocols.
3. **Fault Tolerance**:
   - The single-machine implementation has no fault tolerance. A crash or failure leads to downtime and potential data loss.
   - The Raft-based system is resilient to failures of up to 2 nodes out of 5. It ensures the system remains operational as long as a majority of nodes are available.

“‘yaml

Conclusion

- **Single-Machine Implementation**: Offers low latency and simplicity but lacks fault tolerance and scalability.
- **Replicated System (Raft)**: Provides strong consistency and fault tolerance but incurs higher latency and complexity due to consensus overhead.

“‘yaml

[questions/Consistency_and_Replication/two_phase_commit_safety_liveness](#two_phase_commit_safety_liveness)

Question: *Consider the Two-Phase Commit (2PC) protocol. Which problem does it solve? Does the protocol guarantee safety (correctness) and liveness (does not block)? If not, present a situation in which the protocol might lead to incorrect results.*

“‘yaml

Answer

1. **Problem Solved by Two-Phase Commit (2PC) Protocol**

- **Consistency Across Distributed Systems**:
  - The 2PC protocol is used to ensure **atomicity** in distributed transactions, meaning either all operations across distributed nodes are committed, or none are, ensuring a consistent state across the system.
  - It is particularly useful when multiple nodes participate in a transaction that requires agreement to commit or abort.

2. **Phases of the Protocol**

- **Phase 1: Prepare Phase**
  - The **coordinator** sends a “prepare” request to all participating nodes.
  - Each node checks if it can commit the transaction. If yes, it votes “yes” and prepares to commit; otherwise, it votes “no.”
- **Phase 2: Commit/Abort Phase**
  - If all nodes vote “yes,” the coordinator sends a “commit” request, and all nodes commit the transaction.
  - If any node votes “no,” the coordinator sends an “abort” request, and all nodes abort the transaction.

“‘yaml

3. **Safety (Correctness)**

- **Guaranteed**:
  - The 2PC protocol ensures **safety** because:

    * Nodes commit only after receiving a definitive decision (commit or abort) from the coordinator.
    * Once a node commits, it does not roll back, maintaining a consistent global state.

4. **Liveness (Does Not Block)**

- **Not Guaranteed**:
  - 2PC can block under certain conditions, as it relies on the availability of the coordinator.
  - If the coordinator fails during the commit phase (after some nodes commit and others are undecided), the undecided nodes cannot proceed, resulting in blocking.

“‘yaml

5. **Situation Where 2PC Fails to Guarantee Liveness**

- **Scenario**: Coordinator Failure During Commit Phase
  1. A transaction is initiated, and all participating nodes vote “yes” during the prepare phase.
  2. The coordinator sends a “commit” message to some nodes but fails (e.g., due to a crash) before informing all nodes.
  3. **Result**:
     - Nodes that received the “commit” message commit the transaction.
     - Nodes that did not receive the “commit” message remain in the prepared state, waiting indefinitely for the coordinator’s decision.
- **Impact**:
  - The system is now in an inconsistent state: some nodes have committed, and others are undecided, leading to a **blocked state** for the undecided nodes.

“‘yaml

6. **Conclusion**

- The 2PC protocol solves the problem of ensuring atomicity and consistency in distributed transactions.
- While it guarantees **safety** by preventing inconsistent states, it does **not guarantee liveness**, as a failure in the coordinator can lead to blocking situations.
- This limitation highlights the trade-offs in distributed systems and has led to the development of advanced protocols like **Three-Phase Commit (3PC)** to address the blocking issue.

“‘yaml

[questions/Fault_Tolerance/Consensus_with_Process_Failures](#questions/Fault_Consensus_with_Process_Failures)

Question: *Consider N servers that receive reimbursement requests. Periodically they have to agree on the total number of reimbursements accepted. Suppose the system is synchronous and communication is reliable. Is it possible to reach consensus if processes may fail? How many non-faulty processes are needed to reach such consensus? Which distributed algorithm among those you studied would you use to reach it? Provide a brief description of the algorithm.*

“‘yaml

Answer

1. Is Consensus Possible in a Synchronous System with Process Failures? Yes, consensus is possible in a synchronous system with reliable communication, even when some processes may fail. The key is to ensure that there are enough non-faulty processes to make decisions collectively, despite some failures.

“‘yaml

2. Number of Non-Faulty Processes Needed

To achieve consensus in a synchronous system with **crash failures** (where processes may stop functioning but do not behave maliciously), we need: \[N \geq f + 1 \] where \(f \) is the maximum number of processes that can fail. This requirement ensures that there are always enough non-faulty processes to reach a decision, even in the worst-case scenario of \(f \) failures.

“‘yaml

3. Distributed Algorithm: Simple Majority Agreement Algorithm

Since we are considering a synchronous system, a simple agreement algorithm can be employed. Here's how it works:

Description of the Simple Majority Agreement Algorithm

1. **Initial Setup**: Each process starts with its initial value, which, in this case, is the count of reimbursement requests it has accepted.
2. **Communication Round**:
   - Each process sends its current value to every other process. Given the synchronous nature of the system, all messages are guaranteed to be received within a known time frame.
3. **Aggregation Phase**:
   - Each process collects the values from all other processes, including its own. It then computes the total number of reimbursements by summing up these values.

4. **Decision Phase**:
    - Each non-faulty process uses the aggregated total to reach a consensus on the number of accepted reimbursements. Since all processes perform this step in the same way and have the same set of values (from reliable communication), they all agree on the same result.

“‘yaml

4. Assumptions for the Algorithm to Work

1. **Synchronous System**: The algorithm relies on the assumption that all messages are delivered within a known, bounded time frame.
2. **Reliable Communication**: Messages between processes cannot be lost or delayed beyond the defined bounds.
3. **Crash Failures Only**: The algorithm can tolerate processes that crash and stop working but not processes that act maliciously or send incorrect information.

“‘yaml

5. Comparison with a Centralized Server Approach

1. **Centralized Server Approach**:
    - **Description**: A single server gathers all values, computes the total, and then broadcasts the result to all processes.
    - **Traffic Generated**: Every process sends its value to the central server, which then sends the total back to all processes. This results in \(2N \) messages.
    - **Assumptions**: The central server must be highly reliable and must not crash, as it is a single point of failure.
2. **Simple Majority Agreement Algorithm**:
    - **Traffic Generated**: Each process sends its value to every other process, resulting in \(N \times (N - 1) \) messages in total. While this is more communication overhead, it distributes the load evenly and does not have a single point of failure.
    - **Assumptions**: The system can tolerate up to \(f \) crash failures as long as \(N \geq f + 1 \).

“‘yaml

Conclusion

In a synchronous distributed system with reliable communication and potential crash failures, a **Simple Majority Agreement Algorithm** ensures that consensus is reached. This algorithm relies on each process communicating with all others, aggregating values, and agreeing on a total, without needing a centralized server. It is straightforward, robust in the presence of crash failures, and ensures all non-faulty processes reach the same decision. “‘yaml

[questions/Fault_Tolerance/FloodSet_Algorithm_for_Consensus](#questions/Fau FloodSet_Algorithm_for_Consensus)

Question: *Describe the FloodSet algorithm. Which problems does it solve, and under which assumptions? Why are those assumptions fundamental?*

“‘yaml

Answer

1. FloodSet Algorithm: Overview The **FloodSet algorithm** is designed to solve the consensus problem in a distributed system where processes can fail. The algorithm ensures that all non-faulty processes agree on a single value, even in the presence of up to \(f \) crash failures. The key idea is to repeatedly share values among all processes until consensus can be safely achieved.

“‘yaml

2. Problems Addressed

The FloodSet algorithm addresses the **consensus problem** in a distributed system, specifically ensuring: - **Agreement**: All non-faulty processes must decide on the same value. - **Validity**: If all non-faulty processes start with the same initial value, that value must be the decision. - **Termination**: All non-faulty processes must eventually make a decision.

The algorithm ensures that even if some processes crash during execution, the remaining processes can still reach a consistent decision.

“‘yaml

3. How the FloodSet Algorithm Works

1. **Initialization**:
   - Each process \(P_i \) maintains a set \(W_i \) initialized with its own start value.
2. **Execution Rounds**:
   - The algorithm runs for \(f + 1 \) rounds, where \(f \) is the maximum number of processes that can fail.
   - **In Each Round**:
     1. Each process sends its set \(W_i \) to all other processes.
     2. Upon receiving sets from other processes, \(P_i \) adds the received values to its own set \(z‘ W_i \).
3. **Decision**:
   - After \(f + 1 \) rounds, each process makes a decision:
     - If \(|W_i| = 1 \) (i.e., the set contains only one value), the process decides on that value.

– If \(|W_i| > 1 \), the process uses a pre-specified rule (e.g., choosing the maximum value or a default value \(v_0 \)) to make a decision.

“‘yaml

4. Assumptions and Their Importance

1. **Synchronous System**:
   - The algorithm assumes a **synchronous system**, meaning that there are known bounds on message delivery times and process execution speeds.
   - **Why This Is Fundamental**: The assumption of synchrony is crucial because it ensures that all messages sent in a round are received before the next round begins. Without this guarantee, processes might make decisions based on incomplete information, leading to inconsistencies.
2. **Crash Failures**:
   - The FloodSet algorithm is designed to handle **crash failures**, where processes may stop functioning but do not exhibit arbitrary or malicious behavior (unlike Byzantine failures).
   - **Why This Is Fundamental**: Handling only crash failures simplifies the problem, as the algorithm does not need to account for malicious behavior. In the presence of Byzantine failures, more complex protocols would be required.
3. **Bounded Number of Failures**:
   - The algorithm can tolerate up to \(f \) crashes, meaning it requires at least \(f + 1 \) rounds to ensure that all processes can exchange information sufficiently.
   - **Why This Is Fundamental**: This assumption ensures that there are enough rounds for the non-faulty processes to gather all necessary values and make a consistent decision. If more than \(f \) processes fail, the algorithm cannot guarantee consensus.

“‘yaml

5. Example Walkthrough

Consider a scenario with 4 processes (\(P_1, P_2, P_3, P_4 \)), each starting with different initial values. Suppose the system can tolerate up to 1 crash failure (\(f = 1 \)), so the algorithm will run for \(f + 1 = 2 \) rounds: - **Initial Values**: \(W_1 = \{0\}, W_2 = \{1\}, W_3 = \{0\}, W_4 = \{0\} \). - **Round 1**: Each process sends its set to all others. After collecting and merging values: - \(W_1 = \{0, 1\}, W_2 = \{0, 1\}, W_3 = \{0, 1\}, W_4 = \{0, 1\} \). - **Round 2**: The same sets are exchanged, confirming the values. - **Decision**: All processes have \(W = \{0, 1\} \). If using the rule \(\max(W) \), the agreed-upon value would be 1.

“‘yaml

6. Optimizations and Complexities

- The FloodSet algorithm can be optimized by broadcasting the set \(W \) only when new values are learned, reducing communication overhead.
- The algorithm's complexity arises from the need to handle up to \(f \) failures, ensuring all processes can reach agreement through repeated rounds of communication.

“‘yaml

7. Handling Byzantine Failures

If processes can exhibit **Byzantine behavior** (arbitrary or malicious actions), the problem becomes significantly harder: - The simple \(f + 1 \) round approach is not sufficient because Byzantine processes can send conflicting or misleading information. - To handle Byzantine failures, more complex protocols, such as **Lamport's Byzantine Generals Algorithm**, are required. These protocols often need at least \(3f + 1 \) total processes to tolerate \(f \) Byzantine failures.

“‘yaml

Conclusion

The **FloodSet algorithm** provides a robust way to achieve consensus in a synchronous distributed system with crash failures. It ensures that all non-faulty processes reach the same decision despite up to \(f \) crashes, given the assumptions of synchronous communication and bounded failures. These assumptions are fundamental to the algorithm's correctness, as they provide the necessary guarantees for message delivery and process coordination. “‘yaml

[questions/Fault_Tolerance/Reliable_Group_Communication_with_Process_Fail Reliable_Group_Communication_with_Process_Failures)

Question: *Describe how to efficiently provide reliable group (multicast) communication among a set of processes. How do things change if processes may fail? Explain how the problem changes and how to address the new situation.*

“‘yaml

Answer

1. Efficient Reliable Group Communication Reliable group communication is crucial for maintaining data consistency and ensuring that all group members receive messages correctly, especially in systems that utilize replication for fault tolerance. Here's how it can be efficiently implemented:

1. **Unreliable Multicast Foundation**: Reliable multicast is often built on top of an unreliable multicast protocol (such as UDP). The challenge is to

ensure that all group members receive the message, even if the underlying network is unreliable.

2. **Approaches for Reliable Multicast**:
   - **Positive Acknowledgments (ACKs)**:
     - Each recipient sends an acknowledgment (ACK) to the sender after receiving a message.
     - If the sender does not receive an ACK within a specified time, it retransmits the message.
     - **Drawback**: This approach can lead to an **ACK implosion** in large groups, where the sender is overwhelmed by numerous ACKs.
   - **Negative Acknowledgments (NACKs)**:
     - Recipients only send a NACK if they detect that a message is missing.
     - The sender retransmits the missing message upon receiving a NACK.
     - To manage potential **NACK implosion**, recipients use a timeout and wait before broadcasting a NACK. If one process broadcasts a NACK, others cancel their own NACKs to minimize traffic.
3. **Scalable Techniques**:
   - **Hierarchical Feedback Control**:
     - The group is divided into subgroups, each with a coordinator. The coordinators form a hierarchy with the sender as the root.
     - Coordinators manage retransmissions within their subgroups and communicate with higher-level coordinators as needed.
     - This reduces the number of messages sent to the sender and distributes the communication load.

“‘yaml

2. Challenges When Processes May Fail

When processes can fail, the reliability of group communication becomes more complex. Failures introduce uncertainties about whether messages have been delivered, and group membership can change dynamically.

Problem Changes:

1. **Uncertainty**: If a process fails while sending a message, some members might have received the message while others have not.
2. **Membership Changes**: If processes join or leave the group (including crashes), it complicates the delivery guarantees and ordering of messages.

“‘yaml

3. Addressing Process Failures

To handle these challenges, the concept of **Virtual Synchrony** is used. Virtual synchrony ensures that messages and membership changes are consistently managed across all processes.

Virtual Synchrony Model:

1. **Basic Principles**:
   - **Message Stability**: Messages are only considered stable when all members of the current group have received them. Until then, messages are buffered and may be retransmitted.
   - **View Changes**: A view change occurs when a process joins or leaves the group. During a view change, processes stop sending new messages and focus on ensuring all pending messages are either delivered to all or discarded.
   - **Epochs**: Communication happens in epochs separated by view changes. Messages must be delivered either before or after a view change, never in between.
2. **Implementation Steps**:
   - When a process detects a membership change (e.g., a crash), it initiates a view change.
   - All processes multicast their pending unstable messages and then send a **flush message** to indicate they are ready for the new view.
   - Once a process receives a flush message from all other members, it installs the new view and resumes normal operations.
   - If a message sender crashes, its message is either delivered to all or discarded, ensuring consistency.

Atomic Multicast:

- In systems where it's critical that all messages are delivered in the same order to all group members, **atomic multicast** is used. This ensures totally-ordered delivery and builds on virtual synchrony to handle failures gracefully.

“‘yaml

Conclusion

Efficient reliable group communication can be achieved using positive or negative acknowledgment strategies, combined with techniques like hierarchical feedback control. When processes can fail, the challenge is to ensure consistent message delivery and manage dynamic group membership. Virtual synchrony addresses this by coordinating message delivery and view changes, ensuring that all non-faulty processes have a consistent view of the system. “‘yaml

[questions/Fault_Tolerance/Types_of_Failures_in_Distributed_Systems](#quest Types_of_Failures_in_Distributed_Systems)

Question: *Describe and classify the type of failures that may happen in a distributed system.*

“‘yaml

Answer

In distributed systems, both processes and communication channels are susceptible to various types of failures. These failures are classified to better understand and manage their impact on the system's behavior. The failure model breaks down into several categories:

“‘yaml

1. **Omission Failures**

- **Description**: Omission failures occur when an expected action is not performed, either by a process or a communication channel.
- **Types**:
  - **Process Omission**: The process fails or crashes, making it unable to execute further tasks.
  - **Channel Omission**: Failures that occur when messages are lost. These can be further categorized into:
    - **Send Omission**: A message is lost while traversing the sender's operating system or network stack.
    - **Channel Omission**: A message is lost while traversing the physical network.
    - **Receive Omission**: A message is lost when passing through the receiver's network stack or operating system.
- **Example**: A packet gets dropped due to network congestion, or a server crashes and stops processing requests.

“‘yaml

2. **Byzantine (or Arbitrary) Failures**

- **Description**: These are the most complex and unpredictable failures. In a Byzantine failure, processes or communication channels can behave erratically, such as performing unintended actions or delivering incorrect information.
- **Types**:
  - **Process Byzantine Failures**: A process may omit necessary steps in its computation or introduce unexpected, erroneous behavior.
  - **Channel Byzantine Failures**: The message content may be corrupted, non-existent messages may be delivered, or real messages may

be duplicated and delivered multiple times.
- **Example**: A compromised node in a network sends conflicting information to different parts of the system, or a communication channel delivers corrupted messages that mislead the recipients.

"'yaml

3. Timing Failures

- **Description**: Timing failures occur when a system does not meet the predefined time constraints, and they are specific to synchronous systems.
- **Types**:
  - **General Timing Failures**: When a process or message takes longer to complete than the time limit set for the system. This violation can disrupt real-time operations or deadlines.
- **Example**: A service expected to respond within 5 milliseconds takes 50 milliseconds, potentially causing downstream systems to malfunction or timeout.

"'yaml

Failure Detection in Synchronous vs. Asynchronous Systems

- **Synchronous Systems**: Detecting failures is straightforward because the system operates within strict time constraints. If a process or communication does not complete within a set time limit, it is assumed to have failed. For example, in "Pepperland," if one division does not receive a periodic message from another within a specified timeframe, it concludes the other division has been defeated.
- **Asynchronous Systems**: Detecting failures is more complicated because there are no strict time constraints. A delayed message could mean a failure or simply a delay in message delivery. In "Pepperland," it is difficult to determine whether the other division is defeated or if the messenger is just delayed while crossing the valley.

"'yaml

Conclusion

Understanding the types of failures in a distributed system is crucial for designing mechanisms that can tolerate or recover from such issues. **Omission failures** are more common for processes, while **Byzantine failures** frequently occur in communication channels. **Timing failures** are relevant only in synchronous systems and can be difficult to handle in asynchronous environments. Each failure type requires a tailored approach for detection and recovery.

"'yaml

[questions/Modeling/Mobile_Code_Paradigms](#questions/Modeling-Mobile_Code_Paradigms)

Question: Describe mobile code paradigms with examples.

Answer:

Overview of Mobile Code:

Mobile code refers to the ability to relocate components of a distributed application during runtime. This architecture provides flexibility and adaptability in distributed systems by enabling the movement of code and sometimes the state of execution across different hosts. There are different paradigms depending on where the code originates and where it executes.

“‘yaml

Mobile Code Paradigms:

1. **Client-Server Paradigm:**
   - **Description:** In the client-server model, both the code and execution capabilities are located on the server side. The client simply sends requests and receives processed data or results from the server.
   - **Example:** A web application where the client sends data to a server, which processes it and sends back the results. For instance, a database query handled entirely on the server side.

“‘yaml

2. **Remote Evaluation:**
   - **Description:** In remote evaluation, the client provides the code, which is then executed on a remote server. The server sends back the evaluated result. This paradigm is powerful but poses security risks because the server executes potentially untrusted code.
   - **Example:** PostScript printing is a classic example, where a printer server receives a PostScript file (code) from a client, executes it to render the document, and prints it.
   - **Security Concerns:** The server must be protected from malicious code, as it has to execute external instructions.

“‘yaml

3. **Code on Demand:**
   - **Description:** In this model, the client requests and downloads code from a server and then executes it locally. The client must be equipped to run the code. This paradigm is common in web technologies.

- **Example:** JavaScript in web browsers. A webpage downloads JavaScript code from a server, which is then executed on the client side to enhance interactivity or provide functionality.
- **Key Consideration:** Clients must have a secure environment to execute the code, as untrusted or poorly written code can be exploited.

“‘yaml

4. **Mobile Agent:**

- **Description:** A mobile agent carries both the code and data. It travels between different hosts to complete execution tasks. The agent can perform partial computations and move to another host to continue or complete the task.
- **Example:** There are no widespread practical examples of mobile agents, but they are theoretically useful in scenarios like distributed information retrieval, where the agent collects and processes data across various locations.
- **Advantages:** Mobile agents can reduce network load by processing data close to its source.

“‘yaml

Mobile Code Technologies:

- **Strong Mobility:**
  Refers to systems that can move both the code and the execution state to another environment. An example is a mobile agent that carries its state and resumes execution seamlessly on another host. Few systems support this due to complexity.
- **Weak Mobility:**
  Involves moving code without preserving the execution state. Commonly used in platforms like Java, .Net, and the Web. Examples include remote evaluation and code on demand.

“‘yaml

Advantages of Mobile Code:

- **Flexibility:** Code components can be updated at runtime without halting the system.
- **Enrich Functionality:** Existing components can easily be extended or modified.
- **Service Adaptability:** Services can be customized dynamically to meet specific client requirements.
- **Ease of Adding Services:** New services can be integrated seamlessly.

“‘yaml

Challenges:

- **Security Risks:** Executing mobile code safely is complex. The risk of malicious code exploiting vulnerabilities is a significant concern.

“‘yaml

[questions/Naming/Approaches_to_Flat_Naming_LAN_WAN](#questions/Nam Approaches_to_Flat_Naming_LAN_WAN)

Question: Describe various approaches to implement a flat naming system in a LAN and a large-scale WAN.

Answer:

Flat Naming System Overview:

In a flat naming system, names are unique identifiers without any hierarchical structure, making name resolution more challenging as the scale of the network increases. The strategies for implementing flat naming systems differ between a Local Area Network (LAN) and a Wide Area Network (WAN) because of variations in scale, latency, and reliability.

“‘yaml

Approaches in a Local Area Network (LAN):

1. Broadcast-Based Resolution: In smaller networks like a LAN, a common approach is to use broadcasting to resolve names. When a name needs to be resolved, a broadcast message is sent to all nodes in the network, asking who owns the name. The node with the matching identifier replies.

Advantages: - Simple and efficient for small networks with limited nodes.

Disadvantages: - As the number of nodes increases, broadcast traffic can overwhelm the network and reduce efficiency.

“‘yaml

2. Centralized Naming Server: Another approach is to have a single centralized server that maintains a mapping of names to addresses. All name resolution requests are sent to this server.

Advantages: - Easy to implement and manage, with fast response times in a LAN environment.

Disadvantages: - Single point of failure, and performance can degrade if the server becomes overloaded.

“‘yaml

Approaches in a Large-Scale Wide Area Network (WAN):

1. Distributed Hash Table (DHT): In a WAN, a DHT-based system can be used to distribute the name-to-address mappings across multiple nodes. Names are hashed to keys, and the DHT ensures efficient lookup operations by distributing keys evenly among participating nodes.

Advantages: - Scalability and fault tolerance, as the system can handle large numbers of names and nodes distributed across a wide geographic area.

Disadvantages: - Complexity in handling network partitioning and ensuring data consistency, as well as difficulties in optimizing for locality.

“‘yaml

2. Hierarchical Partitioning: Although the naming system remains flat, a hierarchical partitioning of the namespace can be used to distribute the load. The namespace is divided into partitions managed by different servers, which can then communicate for name resolution.

Advantages: - Balances load and improves performance by reducing the number of servers each request needs to query.

Disadvantages: - Partition management can be complex, especially when handling dynamic changes in the network.

“‘yaml

3. Name Caches and Replication: To improve performance in a WAN, caching resolved names locally on nodes is common. Frequently accessed names are stored temporarily, reducing the need for repeated queries across the network. Additionally, replication of name data across multiple servers ensures reliability and quicker access.

Advantages: - Reduces latency and improves reliability by replicating name data closer to where it is needed.

Disadvantages: - Cache consistency and replication overhead can become challenging, especially in a highly dynamic WAN environment.

“‘yaml

Conclusion:

Implementing a flat naming system in a LAN is typically simpler, relying on broadcast or centralized servers. In contrast, a WAN requires more sophisticated strategies like DHTs, hierarchical partitioning, or caching to handle the scale and complexity. The choice of approach depends on factors like the size of the network, desired performance, and fault tolerance requirements.

“‘yaml

[questions/Naming/Approaches_to_Remove_Unreferenced_Entities](#questions/Approaches_to_Remove_Unreferenced_Entities)

Question: Describe and compare the various approaches to remove unreferenced entities in distributed systems.

Answer:

Overview of Unreferenced Entity Removal:

In distributed systems, unreferenced entities (like objects) may eventually become unreachable from any node and need removal. This is handled by identifying these objects through graph-based techniques that differentiate between reachable and non-reachable nodes. Distributed garbage collection is complicated by the lack of global knowledge and potential network failures.

“‘yaml

Approaches to Remove Unreferenced Entities:

1. Reference Counting: Every object maintains a reference counter. When an object is created, its counter starts at 1. When references are removed, the counter decreases, and the object is deleted when it hits zero.

Problems: - Requires reliable message delivery (acknowledgments, duplicate detection). - Prone to race conditions when references are passed between processes. - Performance can suffer due to multiple message requirements, especially in large-scale systems.

“‘yaml

2. Weighted Reference Counting: This approach assigns two equal weights (e.g., 128, 128) to an object. Each time a reference is created, the weight is halved and distributed. When the weights become equal again, the object can be collected.

How Weighted Reference Counting Works: - **Initial Assignment:** When an object is created, it is assigned two equal initial weights, for example, 128 and 128. - **Reference Creation:** When a new reference to the object is made, one of the weights (let’s call it the remaining weight) is halved and assigned to the new reference. The remaining weight is also updated, usually halved as well, to ensure proper bookkeeping. - **Reference Removal and Weight Recombination:**
- When references are deleted, the weight that was associated with that reference is returned to the original weight pool of the object. - As references are destroyed and weights are returned, the weights that were distributed during reference creation are “merged” back, eventually restoring balance. - When all references are gone and all weights have been returned, the original two weights become equal again.

Problems: - Only a limited number of references can be created due to the halving mechanism. - Scalability issues arise as reference creation depletes the available weight.

“‘yaml

3. Reference Listing: Instead of counting references, this method tracks the identities of proxies that hold references to the object.

Advantages: - Insertion and deletion are idempotent, making communication more reliable and consistent. - Easier to handle network failures by periodically pinging clients, although this may not scale well. - Used in Java RMI with added mechanisms like leases for network failure management and UDP as the transport layer.

Problems: - Still susceptible to race conditions when copying references. - Leases must be refreshed by the client, which can be problematic for long network failures.

“‘yaml

4. Distributed Mark-and-Sweep: Adapts traditional mark-and-sweep garbage collection for distributed systems. It requires marking accessible objects and sweeping non-reachable ones.

Process: - **Mark Phase:** Objects reachable from a root are marked grey, proxies and objects are recursively marked, and acknowledgments are expected to complete marking. - **Sweep Phase:** Unmarked (white) objects are collected once all are either black (marked) or confirmed white.

Problems: - Requires a stable reachability graph, which is challenging in a dynamic distributed environment. - Potential system-wide blocking if a distributed transaction occurs. - Scalability issues as the process grows with the number of nodes.

“‘yaml

Comparison of Approaches:

- **Reference Counting:** Simple and intuitive but suffers from performance and reliability issues, especially under network failures or high messaging overhead.
- **Weighted Reference Counting:** Attempts to mitigate race conditions but is limited in scalability due to the finite number of references.
- **Reference Listing:** Offers better reliability and resilience to network failures but may still face race conditions and scalability constraints.
- **Distributed Mark-and-Sweep:** More comprehensive but introduces significant overhead, requiring a stable system and potentially blocking

transactions. Scalability is inherently poor due to the need for global entity knowledge.

“‘yaml

Conclusion:

Each method has trade-offs. Reference Counting is lightweight but unreliable for large systems. Weighted Reference Counting improves reliability but with limited scalability. Reference Listing offers robustness against network issues but still faces challenges in large-scale environments. Mark-and-Sweep is thorough but complex, making it best suited for smaller or more stable distributed systems.

“‘yaml

[questions/Naming/Flat_vs_Structured_Naming_Hierarchy](#questions/Naming/Flat_vs_Structured_Naming_Hierarchy)

Question: *Describe the difference between a flat naming system implemented with a hierarchy of servers and a structured naming system implemented using a similar hierarchy of servers. Compare the two solutions and explain why the latter one is more efficient.*

Answer:

1. **Flat Naming System**:
   - A flat naming system uses unique names without any hierarchical organization. When implemented with a hierarchy of servers, each name is treated as a random identifier. Name resolution in this system can be inefficient because queries may require contacting multiple servers or broadcasting until the correct resource is found, which is resource-intensive and time-consuming.
2. **Structured Naming System**:
   - In a structured naming system, names are organized hierarchically (like file paths or domain names). This structure is mirrored in the hierarchy of servers, where each server is responsible for a segment of the namespace. The name resolution process is streamlined, as queries can be routed directly to the relevant part of the hierarchy, significantly reducing the search time.

Comparison and Efficiency:

- **Flat Naming System**: Less efficient because lookups can involve multiple server queries or broadcasts, leading to high overhead and poor scalability as the system grows.
- **Structured Naming System**: More efficient, as the hierarchical organization allows for faster lookups by directly narrowing down the search

path. It is also more scalable and easier to manage since changes can be isolated to specific parts of the hierarchy.

Why the Structured System is More Efficient: The structured naming system optimizes lookup by using the hierarchy to minimize the search space. Instead of searching broadly, the system directs queries efficiently, saving time and resources and handling large namespaces better than a flat system. “‘yaml

[questions/Naming/Structured_Naming_with_Server_Hierarchy](#questions/Na Structured_Naming_with_Server_Hierarchy)

Question: *Describe structured naming and its implementation through a hierarchy of servers. Explain also why the same hierarchy of servers doesn't perform well in flat naming.*

“‘yaml

Answer

1. Structured Naming

- **Description**: Structured naming organizes names within a **name space**, which is a labeled graph consisting of **leaf nodes** and **directory nodes**.
  - **Leaf Nodes**: Represent named entities, holding information such as identifiers or addresses.
  - **Directory Nodes**: Act as intermediaries with labeled edges pointing to other nodes, organizing the structure in a way that represents relationships between entities.
- **Path Names**: Resources are referred to using path names, like `/alpha/beta/gamma`. These can be absolute or relative, allowing flexible ways to reference resources.
- **Links**:
  - **Hard Links**: Multiple paths can refer to the same entity.
  - **Symbolic Links**: Leaf nodes may store the absolute path names of entities they refer to, instead of their actual addresses.

“‘yaml

2. Implementation Using a Hierarchy of Servers

- **Hierarchical Organization**: In large-scale systems, the name space is distributed across multiple name servers arranged hierarchically:
  - **Global Level**: High-level directory nodes managed by multiple administrations. This layer handles stable and widely distributed names, like top-level domains in DNS.
  - **Administrational Level**: Mid-level directory nodes grouped under separate administrative authorities. This layer manages names within an organization or domain.

  - **Managerial Level**: Low-level directory nodes within a specific administration. This layer handles frequently updated local resources, such as file servers or hostnames.

Example: DNS (Domain Name System)

- DNS is a practical implementation of structured naming. It uses a rooted hierarchical tree structure where:
  - Each subtree represents a domain managed by a specific authority.
  - Name servers are responsible for managing their respective zones.
- **Name Resolution Techniques**:
  - **Iterative Resolution**: The client queries each server in sequence, with servers providing pointers to the next node. This method uses caching to speed up repeated queries.
  - **Recursive Resolution**: The initial server handles the entire resolution process by forwarding requests down the hierarchy until the final result is found. This method reduces communication cost but increases the load on name servers.

“‘yaml

3. Why Hierarchical Servers Perform Poorly with Flat Naming

- **Flat Naming System**: In contrast to structured naming, a flat naming system uses unique identifiers without any hierarchical organization or structure. Names are not organized in directories or paths, and there is no logical relationship between them.
- **Challenges with a Hierarchical Server Setup**:
  - **Inefficient Search**: The hierarchical organization of servers relies on the name space's structured relationships. With flat names, there are no directory paths to guide the search process efficiently. Each lookup may become a broad and exhaustive search across multiple servers, causing significant performance issues.
  - **Lack of Caching Optimization**: The caching mechanisms designed for hierarchical names (e.g., DNS caching) become ineffective. Since flat names do not share common paths or groupings, caching results are less likely to be reused, leading to repeated lookups and higher latency.
  - **No Clear Partitioning**: Hierarchical servers partition the name space based on the directory structure, making distribution and management efficient. Flat naming lacks this natural partitioning, forcing a less organized and more resource-intensive distribution strategy.

“‘yaml

Conclusion

Structured naming systems benefit significantly from hierarchical server implementations, as they leverage the organized name space to optimize lookups and caching. In contrast, flat naming systems do not align well with hierarchical servers, resulting in inefficient search processes and ineffective caching. The lack of a structured relationship in flat naming undermines the advantages of hierarchical organization, making it a poor fit for such systems.

“‘yaml

[questions/Peer_to_Peer/Chord_Lookup_Protocol](#questions/Peer_to_Peer-Chord_Lookup_Protocol)

Question: *Describe the Chord lookup protocol. What information does each process store in its routing table? In the worst case, what is the maximum number of hops that a query traverses to find a given item? Motivate your answer.*

“‘yaml

Answer

1. Chord Lookup Protocol Overview The **Chord lookup protocol** is a distributed hash table (DHT) mechanism used for efficiently locating nodes and items in a peer-to-peer (P2P) system. Nodes and keys are organized in a logical ring structure. Each node and item in the system is assigned a unique \(m \)-bit identifier, typically generated by hashing the IP address (for nodes) or the item (for keys).

- **Item Storage**: An item with key \(k \) is stored at the node whose identifier is the smallest \(id \geq k \). This node is called the **successor** of \(k \).

“‘yaml

2. Routing Information Stored

Each node in the Chord ring maintains the following information in its routing table:

1. **Successor**: Every node keeps track of its immediate successor in the ring. This ensures that even with minimal information, the system can maintain the ring structure and perform linear lookups.
2. **Finger Table**: To optimize the lookup process, each node maintains a **finger table** with \(m \) entries, where \(m \) is the number of bits in the identifier space.
    - **Entry \(i \) in the Finger Table**: The \(i \)-th entry of a node \(n \) points to the first node whose identifier is at least \(n + 2^i \) (for \(i = 0 \) to \(m - 1 \)).

- This setup allows a node to forward queries exponentially closer to the desired key, reducing the number of hops needed for a lookup.

“‘yaml

3. Lookup Process

When a node \(n \) receives a query for an item with key \(k \): 1. **Check Locally**: The node first checks if it is responsible for the key \(k \) (i.e., if \(k \) is between the node's identifier and its successor's identifier). 2. **Forward Query**: If the item is not stored locally, the node uses its finger table to forward the query. The query is sent to the largest node in the finger table that does not exceed \(k \), effectively halving the remaining distance to \(k \) in each step.

“‘yaml

4. Maximum Number of Hops (Worst Case)

- In the **worst-case scenario**, a query traverses \(O(\log N) \) hops to find the desired item, where \(N \) is the total number of nodes in the system.
- **Motivation**:
  - Each hop in the lookup process reduces the distance to the target key by approximately half. Since the identifier space has \(N \) nodes and the distance can be halved at each step, the number of hops required is \(\log N \).
  - This logarithmic complexity is a result of the exponential leap provided by the finger table entries, which ensure that each query makes significant progress towards the target node.

“‘yaml

5. Joining New Nodes

When a new node \(n \) joins the system: 1. **Initialization**: The new node \(n \) initializes its predecessor and finger table using information from an existing node \(n_0 \) already in the system. 2. **Updating Existing Nodes**: The predecessor and finger tables of other nodes are updated to reflect the addition of \(n \). This ensures that the lookup structure remains efficient. 3. **Time Complexity**: The joining process typically takes \(O(\logˆ2 N) \) hops with high probability to update all necessary nodes.

“‘yaml

Conclusion

The Chord lookup protocol efficiently handles queries in a distributed system using a **finger table** that allows for logarithmic lookup complexity, \(O(\log N) \), even in the worst case. Each node maintains a routing table of size \(O(\log

N) \), which provides a scalable and efficient means of locating items in a large network. However, the system's performance can be affected by frequent node joins and departures, requiring consistent updates to maintain efficiency. “‘yaml

[questions/Peer_to_Peer/Parameter_Passing_RPC_RMI](#questions/Peer_to_Peer/Parameter_Passing_RPC_RMI)

Question: Consider RPC and RMI and focus on message passing. Describe how parameter passing works in the two systems. Why passing parameters by reference is problematic in RPC, how RMI addresses this, and vice versa why passing by value is problematic for RPC.

Answer:

Parameter Passing in RPC and RMI:

“‘yaml

Remote Procedure Call (RPC):

- **Parameter Passing:** In RPC, parameter passing generally uses *pass-by-value*. When a procedure is called on a remote system, the parameters are serialized into a message, sent over the network, and then deserialized on the remote server. This means only a copy of the data is sent, and any changes made to parameters on the server side are not reflected on the client side.

“‘yaml

Remote Method Invocation (RMI):

- **Parameter Passing:** RMI, used in object-oriented distributed systems (e.g., Java), supports both *pass-by-value* and *pass-by-reference*. Objects can be passed by value if they are serializable, meaning their state is copied and transmitted. Alternatively, *remote objects* can be passed by reference, allowing the client to invoke methods on the remote object as though it were local.

“‘yaml

Why Passing Parameters by Reference is Problematic in RPC:

- **Issue:** RPC aims to make remote calls appear as similar as possible to local calls, but it operates across a network. Passing parameters by reference in a remote context would require the remote server to have direct access to the client's memory space, which is impossible over a network.
- **Implication:** Since the remote procedure cannot access the client's memory directly, *pass-by-reference* doesn't work as it does locally. Instead, RPC

must rely on pass-by-value, which copies the data. However, this prevents the remote function from modifying the original data on the client side.

“‘yaml

How RMI Addresses This:

- **Solution in RMI:** RMI supports passing objects by reference if those objects are *remote objects* (i.e., objects that implement a remote interface). When passed by reference, the client and server interact with the object remotely, allowing method invocations to modify the object. Non-remote objects, however, are passed by value, meaning their state is copied and any changes are local to the remote object.
- **Advantages:** This flexibility in RMI enables better modeling of object-oriented interactions across distributed systems, allowing objects to be interacted with remotely without copying their entire state.

“‘yaml

Why Passing by Value is Problematic for RPC:

- **Issue:** Passing by value in RPC can lead to inefficiencies, especially with large data structures, as entire objects need to be copied and transmitted over the network. Additionally, any updates made to these values on the remote side are not reflected on the client side, potentially causing inconsistency or requiring additional mechanisms for synchronization.
- **Trade-offs:** While copying small amounts of data might be manageable, it becomes a performance bottleneck with large or complex data structures. RPC systems have to balance the overhead of data transfer with the simplicity of not maintaining shared memory states across distributed systems.

“‘yaml

[questions/Peer_to_Peer/Pub_Sub_vs_RPC_and_DHT_Dispatching](#questio Pub_Sub_vs_RPC_and_DHT_Dispatching)

Question: Describe pub/sub in general and the difference with RPC. Then explain how a generic DHT can be used to implement a distributed dispatching service. Which is the main limitation of this approach?

Answer:

Pub/Sub Overview:

Publish/Subscribe (Pub/Sub) is a messaging pattern where senders (publishers) and receivers (subscribers) are decoupled. Publishers send messages without knowing who the subscribers are, and subscribers receive messages without knowing who the publishers are. The communication is managed by an intermediary

(e.g., a message broker) that handles message distribution based on topics or event types.

Key Characteristics: - **Decoupling:** Publishers and subscribers are not aware of each other, providing flexibility and scalability. - **Asynchronous Communication:** Messages are sent and received asynchronously, promoting loose coupling in distributed systems.

“‘yaml

Difference Between Pub/Sub and RPC:

RPC (Remote Procedure Call):

- **Description:** In RPC, a client directly calls a method on a remote server, expecting a response. The communication is synchronous, and the caller needs to know the address of the remote service.
- **Tight Coupling:** The client and server are tightly coupled because the client must be aware of the server's interface and address.

Pub/Sub:

- **Description:** The communication is indirect and event-driven. Publishers do not wait for responses from subscribers, and the system can handle many-to-many communication more efficiently.
- **Loose Coupling:** Pub/Sub decouples components, allowing them to operate independently.

Summary: The main difference lies in the communication model: RPC is synchronous and tightly coupled, while Pub/Sub is asynchronous and loosely coupled.

“‘yaml

Using a Generic DHT for Distributed Dispatching:

Distributed Hash Table (DHT): A DHT is a decentralized structure that maps keys to values across multiple nodes. It provides efficient lookup operations and is commonly used in peer-to-peer networks.

Implementing a Distributed Dispatching Service:

- **Mechanism:** In a Pub/Sub system, a generic DHT can be used to store and manage subscriptions. Topics or event types are hashed to specific keys, and the associated subscribers are stored in the nodes responsible for those keys.
- **Message Dispatching:** When a message is published for a particular topic, the DHT is used to efficiently locate the subscribers and forward the message to them.

Advantages: - **Scalability:** The DHT structure distributes the load evenly across nodes, making it scalable and efficient for large systems. - **Efficient Lookups:** The hash-based mechanism ensures that messages can be dispatched quickly to the relevant subscribers.

“‘yaml

Main Limitation of Using DHT for Dispatching:

1. **Lack of Advanced Filtering:** DHTs are primarily designed for key-based lookups and do not support complex queries or content-based filtering. Subscribers can only register for exact topics, limiting the flexibility of the dispatching service.
2. **Event Ordering:** Ensuring message order and consistency in delivery can be challenging due to the distributed nature of DHTs.
3. **Network Dynamics:** DHTs are sensitive to changes in network topology (e.g., nodes joining or leaving), which can impact message reliability and introduce delays or missed messages.

“‘yaml

[questions/Peer_to_Peer/Query_Flooding_vs_Structured_P2P_Search](#questi Query_Flooding_vs_Structured_P2P_Search)

Question: *A P2P system is used to store and retrieve documents. Discuss why the query flooding approach enables searching for documents based on their content, whereas this type of search is not possible within a structured P2P network such as Chord.*

“‘yaml

Answer

1. Query Flooding Approach in Unstructured P2P Networks The **query flooding approach** is commonly used in unstructured P2P networks, such as Gnutella. Here’s how it works: - **Join**: When a new node joins the network, it connects to an “anchor” node and discovers other peers by sending a **PING** message, receiving **PONG** replies, and establishing connections with discovered neighbors. - **Search**: When a node needs to search for a document, it sends a query to its neighbors, which then forward the query to their neighbors, and so on. This process continues until the query reaches a node that has the requested document or until the query’s **HopToLive** counter reaches zero, limiting the query’s spread. - **Fetch**: Once the document’s location is found, it can be fetched directly from the peer that holds it.

Advantages of Query Flooding for Content-Based Search

1. **Flexible Search Capabilities**:

- In an unstructured P2P network, search queries can be based on the **content** of documents, using techniques such as simple text matching, fuzzy text search, or even advanced content-based queries.
- The decentralized and flexible nature of the network means there is no strict structure or predefined key-based lookup mechanism, allowing for arbitrary content searches.

2. **No Need for Structured Indexing**:
   - Since there is no structured index, nodes do not need to maintain a strict mapping of content to locations. Instead, queries can freely propagate through the network, making it possible to search based on document content or even metadata.

“‘yaml

2. Limitations of Structured P2P Networks like Chord Structured P2P networks, such as Chord, organize data and nodes using a well-defined structure, typically based on consistent hashing. Here's how Chord works: - **Data Placement**: Each node and document is assigned a unique identifier using a hash function. The document with key \(k \) is stored at the node whose identifier is the smallest \(id \geq k \) (the document's successor). - **Efficient Lookup**: Chord uses a **finger table** to route queries efficiently, with \(O(\log N) \) hops required in the worst case.

Why Content-Based Search Is Not Possible in Chord

1. **Key-Based Search**:
   - Chord and other structured P2P networks rely on **key-based lookups**. A search query must specify the exact key of the item being sought. The hash function ensures efficient routing but does not support queries based on arbitrary content.
   - As a result, you cannot perform searches based on document content, keywords, or fuzzy matches. You need to know the exact key (or a specific hash) to locate a document.
2. **Deterministic Data Placement**:
   - In Chord, data placement is deterministic and strictly based on the hash of the item. This means that documents are distributed in a way that does not facilitate content-based searching.
   - Without a global or distributed indexing mechanism that supports content-based queries, Chord cannot perform searches based on the internal contents of the documents.

“‘yaml

3. Summary of Differences

- **Query Flooding (Unstructured P2P)**:
  - **Pros**:

        * Supports flexible and content-based searches.
        * No need for a predefined data structure or hash-based routing.
    - **Cons**:
        * Inefficient due to the "flood" of requests, which can lead to high network traffic.
        * Search scope and stability can be problematic in large or highly dynamic networks.
- **Chord (Structured P2P)**:
    - **Pros**:
        * Efficient, scalable lookups using \(O(\log N) \) hops.
        * Deterministic data placement ensures consistency and fast query resolution for key-based searches.
    - **Cons**:
        * Only supports key-based lookups, making content-based searches impossible without additional indexing mechanisms.
        * Less flexible for arbitrary or complex search queries.

"'yaml

Conclusion

The **query flooding approach** in unstructured P2P networks is suitable for content-based searches because it allows queries to be propagated widely and flexibly without the need for structured indexing. However, this comes at the cost of inefficiency and potential network congestion. On the other hand, structured P2P networks like Chord provide efficient key-based lookups but cannot natively support content-based searches due to their reliance on a strict hash-based data organization. "'yaml

[questions/Peer_to_Peer/p2p_query_flooding_vs_chord](#questions/Peer_to_P2p_query_flooding_vs_chord)

Question: *A P2P system is used to store and retrieve documents. Discuss why the query flooding approach enables searching for documents based on their content, whereas this type of search is not possible within a structured P2P network such as Chord.*

"'yaml

Answer

1. Query Flooding in Unstructured P2P Systems (e.g., Gnutella)

- **How Query Flooding Works**:
    - In an unstructured P2P system like Gnutella, nodes are connected in a random, decentralized topology.
    - When a query is initiated, it is **flooded to all neighboring nodes**, and these nodes recursively forward the query to their own neighbors.

  - To prevent infinite propagation, the query message includes a **HopsToLive (TTL)** field, which decrements with each hop and stops the query after a certain limit.
- **Content-Based Searching**:
  - Nodes in Gnutella do not organize or store data based on predefined keys or addresses.
  - Instead, queries can search for arbitrary content (e.g., keywords or phrases in a document) because flooding reaches multiple nodes, each capable of independently checking the query against its locally stored content.
- **Advantages of Query Flooding**:
  - **Dynamic Discovery**: No need to publish or pre-index documents; nodes can dynamically respond to queries that match their stored content.
  - **Flexibility**: Enables searches for content without requiring an exact match for keys or addresses.
  - **Decentralization**: No reliance on a central authority or predefined mapping.

“‘yaml

2. Why Content-Based Search is Not Possible in Structured P2P Systems (e.g., Chord)

- **Structured Topology**:
  - In Chord, nodes are organized in a logical ring structure, with each node responsible for a specific range of keys derived from a consistent hashing function.
  - Data (e.g., documents) is stored and retrieved based on these **keys**, which are exact hashes of the document's name or identifier.
- **Key-Based Searching**:
  - Searching in Chord relies on **exact key lookups**. For instance, to retrieve a document, the query must specify the exact hash of the document's name.
  - Arbitrary content-based searches (e.g., searching for a keyword within a document) are not possible, as the system only knows the location of data based on its key.
- **Routing Efficiency but Limited Flexibility**:
  - Chord efficiently routes queries to the correct node in **O(log N)** hops, but this efficiency comes at the cost of flexibility.
  - Without flooding, the system cannot dynamically query multiple nodes to match arbitrary content.

“‘yaml

3. Comparison of Query Flooding and Structured P2P (Chord)

Aspect	Query Flooding (Unstructured)	Structured P2P (Chord)
Search Mechanism	Floods queries to all nodes, reaching many	Routes queries to specific nodes via DHT
Search Type	Content-based (e.g., keywords)	Key-based (e.g., hashed identifiers)
Topology	Random, decentralized	Structured, organized (e.g., DHT ring)
Efficiency	High network overhead, not scalable	Scalable and efficient (O(log N) routing)
Flexibility	Supports dynamic, arbitrary searches	Restricted to exact key lookups

“‘yaml

4. Conclusion Query flooding in unstructured P2P networks enables dynamic, content-based searching because it does not rely on predefined keys or structure. In contrast, structured P2P systems like Chord optimize for efficiency and scalability but limit searches to exact key matches, making content-based discovery impossible.

“‘yaml

[questions/Streaming_Service_IP_Protocol_Limitations](#questions-Streaming_Service_IP_Protocol_Limitations)

Question: *You want to implement your own video streaming service. Describe the specific requirements of a similar service that do not fit the characteristics of the IP protocol and the mechanisms you could put in place to address those limitations.*

“‘yaml >I couldn’t find something in the textbook that answers this question. So, this answer is provided by chatgpt and might not align with the curriculum ### Answer #### 1. **Specific Requirements of a Video Streaming Service**

When designing a video streaming service, there are several critical requirements that do not align well with the inherent characteristics of the IP protocol:

1. **Low Latency**:
   - **Requirement**: Streaming must have minimal delays to provide a smooth user experience, avoiding buffering and lag.
   - **IP Protocol Limitation**: The IP protocol uses a best-effort delivery model, which does not prioritize low latency. Packet delays are common and can vary significantly.
2. **Reliable Data Transfer**:

- **Requirement**: Video data should be delivered reliably, ensuring minimal frame drops and high-quality playback.
- **IP Protocol Limitation**: IP does not guarantee the delivery or the order of packets. Packet loss and out-of-order delivery can result in missing frames or choppy video.

3. **Efficient Bandwidth Management**:
   - **Requirement**: Streaming large video files requires efficient use of available bandwidth to maintain video quality and prevent network congestion.
   - **IP Protocol Limitation**: IP does not have built-in mechanisms for congestion control or bandwidth optimization, which are necessary for high-quality streaming.
4. **Scalability**:
   - **Requirement**: The service should be able to handle a large number of concurrent users, distributing video content efficiently to minimize load and prevent service degradation.
   - **IP Protocol Limitation**: IP alone cannot efficiently handle large-scale data distribution, and it lacks built-in load balancing or distribution capabilities.

“‘yaml

2. Mechanisms to Address IP Protocol Limitations

To implement a robust video streaming service, you can use several techniques and technologies to overcome these limitations:

1. **Content Delivery Networks (CDNs)**
   - **Mechanism**: Use CDNs to cache and distribute video content from servers located strategically across different regions. Users are served content from the closest CDN server, reducing latency and improving speed.
   - **Benefits**:
     - Minimizes latency by reducing the physical distance between the server and the user.
     - Enhances reliability by distributing load and preventing single points of failure.
2. **Adaptive Bitrate Streaming (ABR)**
   - **Mechanism**: Implement adaptive bitrate streaming protocols, such as **HLS (HTTP Live Streaming)** or **MPEG-DASH**, which adjust the quality of the video stream in real-time based on the user's current network conditions.
   - **Benefits**:
     - Optimizes video playback by ensuring smooth streaming even when bandwidth fluctuates.
     - Reduces buffering by dynamically lowering video quality when the network is congested.

3. **Using UDP for Real-Time Streaming**
    - **Mechanism**: Instead of TCP, use **UDP (User Datagram Protocol)**, which allows for faster data transmission since it does not require acknowledgment of each packet. Combine UDP with error correction techniques to manage packet loss.
    - **Benefits**:
        - Reduces latency and provides smoother playback, as UDP is less concerned with retransmission delays.
        - Useful for live streaming or time-sensitive content where a small amount of data loss is acceptable.
4. **Error Correction and Recovery Techniques**
    - **Mechanism**: Use **Forward Error Correction (FEC)** or similar mechanisms to handle packet loss. FEC adds redundant data to the stream, allowing the receiver to reconstruct lost packets without needing retransmissions.
    - **Benefits**:
        - Maintains video quality by preventing noticeable errors in the playback.
        - Reduces the need for retransmission, which can introduce delays.
5. **Load Balancing and Distributed Servers**
    - **Mechanism**: Deploy multiple servers and use load balancers to distribute incoming user requests evenly across the network. Load balancing ensures that no single server is overwhelmed, enhancing service reliability.
    - **Benefits**:
        - Increases scalability and maintains high performance even under heavy loads.
        - Improves fault tolerance by redistributing traffic if a server fails.
6. **Quality of Service (QoS) Mechanisms**
    - **Mechanism**: Implement QoS techniques to prioritize video streaming traffic over less critical data on the network. This ensures that video packets are delivered in a timely manner.
    - **Benefits**:
        - Reduces latency and packet loss for critical video data.
        - Enhances the overall user experience by ensuring high-priority delivery of video streams.

“‘yaml

Conclusion

The IP protocol's limitations, such as unpredictable latency, lack of reliability, and inefficient bandwidth management, can be mitigated using CDNs, adaptive bitrate streaming, UDP, and error correction techniques. These mechanisms ensure high-quality, scalable, and low-latency video streaming, addressing the specific demands of a modern streaming service.

“‘yaml

[questions/Synchronization/Causal_vs_Total_Ordering](#questions/Synchronizat Causal_vs_Total_Ordering)

Difference Between Causal Ordering and Total Ordering

1. Causal Ordering - **Definition**: Causal ordering ensures that messages are delivered in a way that respects the causal relationships between events. In other words, if an event (or message) \(A \) causally precedes another event \(B \) (denoted \(A \rightarrow B \)), then \(A \) must be delivered before \(B \) to all processes. However, events that are independent (concurrent) may be delivered in any order. - **Example**: - If process \(P_1 \) sends message \(A \), and based on receiving \(A \), process \(P_2 \) sends message \(B \), then all processes must deliver \(A \) before \(B \) to respect the causal relationship. - However, if \(A \) and another message \(C \) are independent (no causal relationship), then \(C \) may be delivered before or after \(A \). - **Use Case**: Causal ordering is essential in systems where the logical relationship between events needs to be preserved, such as in collaborative editing or consistent updates to distributed databases.

2. Total Ordering - **Definition**: Total ordering ensures that all messages are delivered in the exact same order to all processes, regardless of any causal relationships. In this case, if one process delivers message \(A \) before \(B \), every other process must also deliver \(A \) before \(B \), even if \(A \) and \(B \) are independent. - **Example**: - If two independent messages \(A \) and \(B \) are sent, and process \(P_1 \) delivers \(A \) first and then \(B \), then every other process in the system must also deliver \(A \) before \(B \). - **Use Case**: Total ordering is required in systems that need a consistent global view of events, such as in distributed databases or in algorithms that require a strict sequencing of events, like consistent replication protocols.

“‘yaml

Key Differences

1. **Respect for Causal Relationships**:
   - **Causal Ordering**: Only enforces the order when events are causally related. It does not enforce any order for independent events.
   - **Total Ordering**: Enforces a strict order for all events, whether or not they are causally related.
2. **Ordering Flexibility**:
   - **Causal Ordering**: More flexible, as it allows different delivery orders for independent events.
   - **Total Ordering**: More strict, as it requires a uniform order for all events across all processes.
3. **Complexity and Overhead**:

- **Causal Ordering**: Generally has lower communication and synchronization overhead, as it only tracks causal dependencies.
- **Total Ordering**: Typically incurs higher overhead because it requires global coordination to ensure that all processes agree on the order of every message.

“‘yaml

Summary

- **Causal Ordering** focuses on preserving the logical dependencies between events, while **Total Ordering** ensures a uniform sequence of events across all processes, regardless of causal relationships.
- In distributed systems, choosing between causal and total ordering depends on the application's consistency requirements and the trade-offs between communication overhead and ordering guarantees. “‘yaml

[questions/Synchronization/Clock_Synchronization_Approaches](#questions/Sync Clock_Synchronization_Approaches)

Question: *Describe and compare various approaches to synchronize node clocks in a distributed system. Now suppose you have to correlate the readings of multiple distributed microphones to identify the source of a sound. Which clock synchronization method would you use?*

Answer

1. Overview of Clock Synchronization in Distributed Systems

- In distributed systems, node clocks need to be synchronized to maintain consistency and correctly order events. However, clocks are not perfect and tend to drift over time, so periodic resynchronization is necessary.
- The goal of clock synchronization can be categorized into two main types:
  - **Accuracy**: Synchronizing all clocks to an accurate external reference.
  - **Agreement**: Ensuring all clocks are in sync with each other.

“‘yaml

2. Clock Synchronization Approaches

1. **GPS-Based Synchronization**:
   - **Description**: Uses signals from GPS satellites to synchronize clocks accurately. Triangulation is used to determine the position and the time is adjusted based on the delay of signals.
   - **Pros**: High accuracy and works well outdoors where GPS signals are available.
   - **Cons**: Not suitable for indoor environments or situations where GPS signals cannot be received.

2. **Cristian's Algorithm**:
   - **Description**: A client periodically requests the current time from a time server. The client adjusts its clock based on the server's time and the estimated round-trip delay.
   - **Challenges**:
     - Time cannot run backward, so adjustments are made gradually.
     - Assumes a consistent network delay, but inaccuracies can arise if this delay varies significantly.
3. **Berkeley Algorithm**:
   - **Description**: An active time server collects time readings from all nodes, computes an average, and sends out adjustments to synchronize the clocks.
   - **Pros**: Useful for environments where no external time source is available. It ensures clocks are synchronized within the system.
   - **Cons**: Less accurate than GPS or NTP and can be disrupted by network failures.
4. **Network Time Protocol (NTP)**:
   - **Description**: Widely used for synchronizing clocks over large networks. NTP uses a hierarchical structure (strata) with servers directly connected to a UTC source at the top.
   - **Mechanisms**: Uses a combination of multicast, procedure-call mode, and symmetric mode for high accuracy.
   - **Accuracy**: Achieves 1ms accuracy on LANs and 1-50ms over the Internet.
   - **Pros**: Highly scalable and reliable; used globally.
   - **Cons**: May not be precise enough for applications needing microsecond-level accuracy.

"'yaml

3. Application: Correlating Readings from Distributed Microphones

- **Scenario**: To identify the source of a sound accurately, the timestamps from distributed microphones must be synchronized as precisely as possible. This ensures the timing differences in sound arrival can be used to triangulate the sound source.

Recommended Clock Synchronization Method: Network Time Protocol (NTP)

- **Reason**: NTP provides a reasonable balance between accuracy and scalability for synchronizing clocks across a large number of nodes. Although it may not offer microsecond precision, it is typically sufficient for sound source localization, provided that small errors are acceptable.
- **Alternative**: If extremely high precision is required and the microphones are outdoors or in GPS-accessible areas, using **GPS-based synchronization** would be more suitable.

I don't trust chatgpt for this one. imo since the nodes need to be synced only amoung themselves Berkely's is sufficent.

"'yaml

Conclusion

Various approaches to clock synchronization offer different trade-offs in terms of accuracy, reliability, and applicability. For most distributed systems, NTP is the go-to solution. However, for specialized applications like sound source localization, the choice depends on the required precision and environmental constraints. For the microphone correlation task, NTP would be effective, but GPS could be considered for higher precision if available.

`yaml`yaml **Question 2:** *Now suppose you have to correlate multiple readings from geographically distributed vibration sensors to determine the origin of an earthquake with a precision of less than 1 km (seismic waves travel at a maximum speed of 10 km/s).* ### Answer 2

For the scenario of correlating multiple readings from geographically distributed vibration sensors to determine the origin of an earthquake with precision better than 1 kilometer (knowing that seismic waves travel at a maximum speed of 10 km/s), an extremely accurate and reliable clock synchronization method is required.

"'yaml

1. Synchronization Requirements for Earthquake Detection

- **Precision Needed**: To achieve a location precision of less than 1 kilometer, the timestamps of the readings from all sensors must be accurate to within 100 milliseconds (0.1 seconds). This is because, at a speed of 10 km/s, even a 0.1-second error translates to a 1 km discrepancy in locating the source.
- **Challenges**:
  - The synchronization method must account for network delays and ensure that all vibration sensors have a highly accurate, synchronized global time.

"'yaml

2. Recommended Clock Synchronization Method: GPS-Based Synchronization

- **Why GPS?**
  - **High Precision**: GPS-based clock synchronization can provide sub-microsecond accuracy, far exceeding the precision requirement of 100 milliseconds. This level of precision ensures that seismic wave

measurements can be accurately correlated across geographically distributed sensors.
  - **Geographic Distribution**: GPS is ideal for synchronization over large distances, as it uses signals from satellites to synchronize clocks globally.
- **How It Works**:
  - Each sensor is equipped with a GPS receiver that continuously synchronizes the local clock with the GPS time. This ensures that all sensors have a common, highly accurate time reference, independent of network delays.
  - **Fallback Mechanism**: If GPS signals are unavailable (e.g., in indoor or underground locations), a backup synchronization method such as Network Time Protocol (NTP) or Precision Time Protocol (PTP) can be used, although with reduced accuracy.

“‘yaml

3. Alternative Method: Network Time Protocol (NTP)

- **When to Use**: NTP can be a backup solution in environments where GPS is not feasible. However, NTP typically provides only millisecond-level accuracy, which might not be sufficient for the 100-millisecond requirement, especially over long distances or under variable network conditions.
- **Limitations of NTP**:
  - **Network Delay Variability**: NTP's accuracy can be affected by network delays, making it less reliable for precision applications like earthquake detection.
  - **Scalability**: For a large-scale sensor network, NTP may introduce inconsistencies if network paths have significant variability.

“‘yaml

4. Assumptions and Considerations

- **Reliable Communication**: The system must ensure that all sensor readings are transmitted and collected in a reliable and timely manner, with mechanisms in place to handle packet loss or delays.
- **Data Aggregation**: A central server or distributed system would collect the timestamped sensor readings and use the precise timing information to triangulate the origin of the seismic waves.

“‘yaml

Conclusion

For determining the origin of an earthquake with high precision, **GPS-based clock synchronization** is the most suitable method due to its ability to provide

global time accuracy within microseconds. This ensures that all sensor readings can be accurately correlated, meeting the stringent timing requirements imposed by the high speed of seismic waves. “‘yaml

[questions/Synchronization/Pessimistic_Timestamp_Ordering_Protocol](#questic Pessimistic_Timestamp_Ordering_Protocol)

File Title: Pessimistic_vs_Optimistic_Timestamp_Ordering_Protocol

“‘yaml

Question: *Describe pessimistic timestamp ordering. Which problem does this protocol address, how does it work? In a crowded system with a lot of requests to manage and a small dataset to access, would you use pessimistic or optimistic timestamp ordering and why? Additionally, consider a system with a few requests per second and a large database. How would your choice change, and why?*

“‘yaml

Answer

1. Pessimistic Timestamp Ordering: Overview Pessimistic timestamp ordering is a concurrency control protocol used in distributed databases to manage the execution order of transactions, ensuring consistency and preventing conflicts.

“‘yaml

2. Problem Addressed

The primary problem that pessimistic timestamp ordering addresses is **data inconsistency** caused by concurrent access to shared data by multiple transactions. In distributed systems, simultaneous operations on a dataset can lead to issues such as **lost updates**, **dirty reads**, or **non-repeatable reads**. This protocol ensures that transactions are executed in a way that maintains a consistent and conflict-free state.

“‘yaml

3. How Pessimistic Timestamp Ordering Works

1. **Timestamps**:
   - Each transaction is assigned a unique **timestamp** at the beginning. This timestamp determines the transaction’s order in the system.
   - Data items are associated with two timestamps:
     - **Read Timestamp**: The latest timestamp of a transaction that successfully read the item.
     - **Write Timestamp**: The latest timestamp of a transaction that successfully wrote to the item.

2. **Execution Rules**:
   - When a transaction attempts to **read** or **write** a data item, the system checks the item's timestamps to decide whether the operation can proceed:
     - **Read Rule**: A transaction \(T \) can read a data item if \(T \)'s timestamp is greater than or equal to the item's write timestamp. If \(T \)'s timestamp is less, the read operation is rejected, and \(T \) is aborted to prevent reading outdated data.
     - **Write Rule**: A transaction \(T \) can write to a data item if \(T \)'s timestamp is greater than the item's read and write timestamps. If \(T \)'s timestamp is less, the write operation is rejected, and \(T \) is aborted to avoid overwriting data that has been accessed by newer transactions.
3. **Abortion and Restart**: If a transaction is aborted due to a timestamp conflict, it is typically restarted with a new timestamp.

"'yaml

4. Pessimistic vs. Optimistic Timestamp Ordering: Scenarios and Recommendations

Scenario 1: Crowded System with a Small Dataset In a **crowded system** with a **high number of requests** and a **small dataset**, conflicts are frequent due to the limited number of data items and the high contention for access.

- **Pessimistic Timestamp Ordering**:
  - **Pros**: Proactively prevents conflicts by ensuring transactions respect the order of timestamps. This reduces data inconsistency and prevents the need for costly rollbacks.
  - **Cons**: High overhead due to frequent checks and preemptive aborts, which may reduce efficiency in a highly contested environment.
- **Optimistic Timestamp Ordering**:
  - **Pros**: Allows more parallelism and reduces overhead by deferring conflict checks until the end of a transaction. However, it may not be effective if conflicts occur frequently.
  - **Cons**: High cost of rolling back transactions if conflicts are frequent, which could negate the performance benefits of the optimistic approach.
- **Recommendation**: **Pessimistic Timestamp Ordering** is the better choice in this case. The high likelihood of conflicts and the small dataset make it necessary to preemptively manage conflicts to maintain consistency and minimize costly rollbacks.

"'yaml

Scenario 2: System with Few Requests and a Large Database In a system with a **low number of requests per second** and a **large dataset**, the chances of transactions conflicting are much lower.

- **Pessimistic Timestamp Ordering**:
  - **Pros**: Provides strong consistency guarantees but may introduce unnecessary overhead in a system with few conflicts.
  - **Cons**: The preemptive nature of this protocol could reduce system efficiency, especially when conflicts are rare.
- **Optimistic Timestamp Ordering**:
  - **Pros**: More suitable in environments with low contention. Transactions are allowed to proceed without checks, and conflicts are resolved only if they occur, reducing the overhead of constant checking.
  - **Cons**: If a conflict does occur, the transaction must be rolled back, but this is a rare event in systems with low request rates and a large dataset.
- **Recommendation**: **Optimistic Timestamp Ordering** is more appropriate here. The low likelihood of conflicts and the large dataset make it more efficient to allow transactions to proceed without checks, benefiting from lower overhead and better overall system performance.

"'yaml

5. Unified Recommendation Based on System Characteristics

- **High Contention, Small Dataset**: Use **Pessimistic Timestamp Ordering** to ensure consistency and minimize performance loss from frequent conflicts.
- **Low Contention, Large Dataset**: Use **Optimistic Timestamp Ordering** to take advantage of lower overhead and improved efficiency, given that conflicts are infrequent.

Conclusion

The choice between pessimistic and optimistic timestamp ordering depends on the system's characteristics. For a crowded system with a small dataset, **pessimistic ordering** is preferred due to the high risk of conflicts. Conversely, in a system with few requests and a large dataset, **optimistic ordering** is more efficient, leveraging the low probability of conflicts to reduce overhead and increase performance. "'yaml

[questions/Synchronization/Scalar_Clocks_Totally_Ordered_Multicast](#questio Scalar_Clocks_Totally_Ordered_Multicast)

Question: *Describe how scalar clocks can be used to implement a totally ordered multicast communication (clarify the assumptions for the protocol to work). Compare this solution with one based on a centralized server in charge of receiving*

messages via point-to-point links and dispatching them to every group member. Focus your comparison on the traffic generated by the two solutions and the assumptions for the two protocols to operate correctly.

“‘yaml

Answer

1. Using Scalar Clocks for Totally Ordered Multicast Scalar clocks, introduced by Lamport, are used to numerically capture the **happens-before** relationship between events in distributed systems. Here's how they help achieve totally ordered multicast communication:

1. **Mechanism**:
   - Each process \(p_i \) maintains a logical scalar clock \(L_i \), initialized to zero.
   - Before sending a message, \(p_i \) increments \(L_i \) and timestamps the message with \(L_i \).
   - When a message is received, the recipient process updates its clock to: \[L_i = \max(L_{\text{msg}}, L_i) + 1 \] where \(L_{\text{msg}} \) is the timestamp of the received message.
   - Messages (including acknowledgments) are sent using multicast and are queued in timestamp order. A message is delivered to the application only when:
     - It is at the front of the queue.
     - All acknowledgments for that message have been received, ensuring that all processes have an updated global view.
2. **Assumptions for the Protocol to Work**:
   - **Reliable Communication**: The system must ensure reliable delivery of messages, with no message loss, duplication, or reordering.
   - **FIFO Links**: The communication channels must preserve the order of message delivery.
   - **Global Agreement**: All processes must receive messages and their acknowledgments to maintain a consistent ordering.
3. **Why Acknowledgments Are Necessary**:
   - Simply using Lamport's ordering without acknowledgments is insufficient because processes do not have a complete global view of all messages. Acknowledgments provide this global view, ensuring that no lower-timestamped message is still in transit.

“‘yaml

2. Centralized Server-Based Solution

An alternative method for achieving totally ordered multicast is to use a centralized server that manages message ordering:

1. **Mechanism**:

- All processes send their messages to a centralized server via point-to-point links.
- The server assigns a global timestamp to each message, ensuring consistent ordering.
- The server then dispatches the messages to all group members, maintaining the total order.

2. **Assumptions for the Protocol to Work**:
   - **Centralized Reliability**: The centralized server must be highly reliable and must not become a single point of failure.
   - **Stable Network Links**: The communication links with the server should be reliable and have minimal delay to avoid creating bottlenecks.

“‘yaml

3. Comparison: Traffic Generated and Assumptions

1. **Traffic Generated**:
   - **Scalar Clock-Based Solution**:
     - **Multicast Traffic**: Each message is sent using multicast, and acknowledgments are broadcast to all group members.
     - **High Traffic**: The need for acknowledgments significantly increases the number of messages exchanged, especially as the group size grows, resulting in considerable network traffic.
   - **Centralized Server-Based Solution**:
     - **Point-to-Point Traffic**: All messages are routed through the central server, creating a high load on the server but reducing the number of broadcasts.
     - **Lower Traffic**: Compared to the scalar clock method, overall network traffic is typically lower because the server handles the ordering, and there is no need for multiple acknowledgments from every process.
2. **Assumptions for Correct Operation**:
   - **Scalar Clock-Based Solution**:
     - Requires reliable multicast and FIFO communication to ensure that messages and acknowledgments maintain the correct order.
     - Suitable for systems that prioritize distributed control and fault tolerance, but with higher communication costs.
   - **Centralized Server-Based Solution**:
     - Assumes that the central server is always available and capable of efficiently handling the load.
     - Introduces a single point of failure; if the server crashes or becomes overwhelmed, the entire system can be affected.

“‘yaml

Conclusion

- **Scalar Clocks**: Offer a decentralized approach with increased traffic due to multicast and acknowledgment messages. They are appropriate for systems needing high fault tolerance and no central point of failure.
- **Centralized Server**: Simplifies message ordering and reduces network traffic but depends heavily on the reliability and performance of the central server.

Each approach has trade-offs in terms of scalability, fault tolerance, and network efficiency, and the choice depends on the specific requirements of the distributed system.

“‘yaml

[questions/Synchronization/Scalar_Clocks_for_Causal_Ordering](#questions/Syn Scalar_Clocks_for_Causal_Ordering)

Question: *Describe how scalar clocks can be used to guarantee causal ordering in a broadcast communication protocol among a set of known processes. If the set of processes might change at runtime, would you still be able to obtain the same results? Motivate your answer and, in case of a positive answer, describe how.*

“‘yaml

Answer

1. Using Scalar Clocks to Guarantee Causal Ordering **Scalar clocks**, introduced by Lamport, can be used to enforce causal ordering in a distributed broadcast communication protocol. The causal ordering relationship, denoted as \(\rightarrow \), ensures that if event \(e \) causally precedes event \(e' \) (i.e., \(e \rightarrow e' \)), then all processes must receive \(e \) before \(e' \).

How Scalar Clocks Work

1. **Clock Management**:
   - Each process \(P_i \) maintains a scalar clock \(C_i \), initialized to zero.
   - The clock is incremented before every broadcast or message send operation: \[C_i = C_i + 1 \]
   - Each message \(m \) sent by \(P_i \) is timestamped with the current value of \(C_i \).
2. **Message Reception**:
   - When process \(P_j \) receives a message \(m \) from \(P_i \) with timestamp \(T_m \), it updates its scalar clock as follows: \[C_j = \max(C_j, T_m) + 1 \]

- The message is only delivered to the application if all messages that causally precede \(m \) (i.e., messages with a smaller timestamp) have already been delivered.

Guaranteeing Causal Ordering

- By using scalar clocks, messages are processed in the order dictated by their timestamps. This ensures that any message that causally precedes another message is delivered first, preserving the causal relationship.
- Since scalar clocks provide a partial ordering of events, they are sufficient for causal ordering in a static set of known processes.

“‘yaml

2. Handling Dynamic Process Membership

If the set of processes might change at runtime (e.g., processes can join or leave the group), ensuring causal ordering becomes more complex but is still achievable with additional mechanisms.

Challenges with Dynamic Membership

- **Clock Adjustment**: When new processes join, they need to be integrated into the causal ordering mechanism without disrupting the existing order of messages.
- **State Transfer**: New processes must be brought up to date with the messages that have already been sent and the current state of scalar clocks.

Solution for Dynamic Membership

1. **Joining a New Process**:
   - When a new process \(P_{new} \) joins the system, it must first synchronize with an existing process to learn the current state of the system, including:
     - The latest scalar clock values.
     - The set of messages that have been broadcast but not yet delivered.
   - \(P_{new} \) initializes its clock to the maximum clock value of all existing processes to ensure it does not violate causal order: \[C_{new} = \max(C_i \text{ for all existing } P_i) \]
   - This initialization ensures that the new process does not deliver messages out of order.
2. **Leaving a Process**:
   - When a process \(P_{leave} \) leaves the system, it must ensure that any messages it has sent are either delivered or acknowledged by all other processes before leaving.
   - The system may use a **flush protocol** to make sure all pending messages are stabilized before removing the process.

3. **Maintaining Causal Order**:
    - The protocol must ensure that all processes have a consistent view of the membership changes. This can be done using a **view change mechanism**, where processes agree on the new set of members before proceeding with regular message delivery.
    - Membership changes themselves are treated as special events that are causally ordered relative to the broadcast messages.

“‘yaml

3. Motivation and Feasibility

- **Positive Answer**: Yes, it is still possible to maintain causal ordering in a system with dynamic membership, but it requires careful management of process joins and leaves.
- **Rationale**: By synchronizing new processes with the current state of the system and ensuring all processes agree on membership changes before proceeding, we can extend the causal ordering guarantees to a dynamic environment.

“‘yaml

Conclusion

Scalar clocks can effectively guarantee causal ordering in a broadcast communication protocol among a static set of known processes. If the set of processes changes at runtime, additional mechanisms like state synchronization, clock initialization, and view change protocols are necessary to maintain causal ordering. These mechanisms ensure that new processes are correctly integrated and that the causal relationship between events is preserved. “‘yaml

[questions/Synchronization/Vector_Clocks_for_Causal_Ordering](#questions/Sy Vector_Clocks_for_Causal_Ordering)

Question: *Describe how vector clocks can be used to guarantee causal ordering in a broadcast communication among a set of known processes. If the set of processes might change at runtime, would you still be able to obtain the same results? How?*

“‘yaml

Answer

1. Using Vector Clocks for Causal Ordering **Vector clocks** are an extension of scalar clocks that can fully capture the causality relationship between events in a distributed system. They are used to ensure that messages are delivered in an order that respects causal relationships, which is crucial for

maintaining consistency in systems like distributed bulletin boards or replicated databases.

How Vector Clocks Work:

1. **Definition and Initialization**:
   - Each process \(P_i \) maintains a **vector clock** \(V_i \), which is an array of size \(N \) (where \(N \) is the number of processes in the system).
   - \(V_i[j] \) represents the number of events that process \(P_i \) knows have occurred at process \(P_j \).
2. **Rules for Updating Vector Clocks**:
   - **Local Event**: When a process \(P_i \) executes a local event, it increments \(V_i[i] \) by 1.
   - **Message Sending**: Before sending a message, \(P_i \) increments \(V_i[i] \) and attaches the entire vector clock \(V_i \) to the message.
   - **Message Receiving**: When \(P_i \) receives a message from \(P_j \) with a timestamp \(t \):
     - \(V_i[k] = \max(V_i[k], t[k]) \) for all \(k \neq i \).
     - Then \(P_i \) increments \(V_i[i] \).

Determining Causality:

- To determine if event \(e \) causally precedes event \(e' \), we check their vector timestamps: \[V(e) < V(e') \iff V(e)[j] \leq V(e')[j] \text{ for all } j \text{ and } V(e) \neq V(e') \]
- If \(V(e) \) and \(V(e') \) are not related in this way, the events are **concurrent** (denoted as \(e \parallel e' \)).

"'yaml

2. Ensuring Causal Delivery

Vector clocks are used to guarantee **causal ordering** in message delivery. Specifically, if event \(e \) causally precedes event \(e' \), then all processes must deliver \(e \) before \(e' \).

- **Implementation**:
  - A message \(m \) is only delivered to the application if all causally preceding messages have already been delivered.
  - This is achieved by checking the vector clock of the incoming message against the process's current vector clock. If the vector clock conditions are satisfied, the message is delivered; otherwise, it is held until the required messages are received.
- **Example**: Consider a message \(MSG \) sent from \(P_1 \) and a reply \(REPLY \) sent from \(P_2 \). To ensure causal delivery, \(REPLY \) must only be delivered after \(MSG \) has been delivered to all processes.

“‘yaml

3. Handling Dynamic Membership (Process Changes at Runtime)

If the set of processes changes at runtime (e.g., new processes join or existing processes leave), maintaining causal ordering using vector clocks becomes more complex but is still possible with some modifications.

Challenges with Dynamic Membership:

1. **Vector Clock Size**: The size of the vector clock depends on the number of processes \(N \). When a new process joins or an existing one leaves, the vector clock structure must be adjusted.
2. **State Synchronization**: Newly joined processes need to synchronize with the current state of the system, which includes updating their vector clock and learning about any pending messages.

Solutions for Dynamic Membership:

1. **Adding a New Process**:
   - When a new process \(P_{new} \) joins the system, all existing processes update their vector clocks to accommodate the new entry, increasing the size of the clock vector by 1.
   - \(P_{new} \) initializes its vector clock with all entries set to 0 and synchronizes with one or more existing processes to learn the current state of the system, including any messages that need to be causally delivered.
   - All processes broadcast an update indicating the new membership, ensuring everyone adjusts their vector clocks consistently.
2. **Handling Process Departures**:
   - If a process \(P_i \) leaves, the vector clock entries associated with \(P_i \) can either be ignored or compacted, depending on the system's design.
   - If compacting, all remaining processes need to synchronize to ensure that the vector clocks are updated uniformly, and causality is preserved.
3. **View Changes**:
   - Implementing **view changes** (similar to virtual synchrony) helps manage dynamic membership. A view change ensures that all processes agree on the membership before resuming normal operations.
   - During a view change, vector clocks are adjusted, and any necessary state synchronization is performed.

“‘yaml

4. Conclusion

Vector clocks effectively guarantee causal ordering among a fixed set of known processes. If the set of processes might change at runtime, the system can still maintain causal ordering by dynamically adjusting the size of the vector clocks and using synchronization mechanisms to integrate new processes or handle departures. While this adds complexity, it ensures that the causal relationships between events are preserved even in a dynamic distributed environment. “‘yaml

[questions/Synchronization/Virtual_Synchrony_Concept_and_Implementation](# Virtual_Synchrony_Concept_and_Implementation)

Question: *Describe the concept of virtual synchrony in general and a possible implementation. Clarify the assumptions required for the protocol to be correct.*

“‘yaml

Answer

1. Concept of Virtual Synchrony Virtual synchrony is a communication model used in distributed systems to ensure that all group members have a consistent view of the sequence of events, even in the presence of failures. This model is especially useful in systems that require coordinated actions, such as fault-tolerant replication and group communication.

The core idea of virtual synchrony is to guarantee that all members of a process group see the same sequence of messages and view changes (e.g., when members join or leave the group). Even if a process crashes or network failures occur, all non-faulty members maintain a consistent state and agree on the messages that were delivered before the group membership changed.

“‘yaml

2. Implementation of Virtual Synchrony

One of the well-known implementations of virtual synchrony is in the ISIS system, which ensures message consistency and reliable communication between group members. Here’s how it works:

1. **Message Transmission and Stability**:
   - Messages are sent using **reliable and FIFO point-to-point channels**, which ensure that messages are delivered in the order they were sent, but do not guarantee that all group members will receive the message if the sender crashes.
   - A message is considered **stable** when it is confirmed that all members of the current group view have received it. Until stability is confirmed, processes buffer the message.
2. **View Changes**:

- A **view change** occurs when the membership of the group changes, such as when a process fails or a new one joins. A distributed component handles view change notifications.
- Upon receiving a view change notification, each process:
  - Stops sending new messages.
  - Multicasts all pending, unstable messages to the remaining members of the old view to ensure everyone has the same set of messages.
  - Marks these messages as stable and sends a **flush message** to indicate that it is ready to install the new view.
- A process can only install the new view after receiving a flush message from every other process in the new view.

3. **Steps in the Protocol**:
   - **Step 1**: A process detects a change (e.g., another process crashes) and initiates a view change.
   - **Step 2**: Each non-faulty process sends its unstable messages to the group and then a flush message.
   - **Step 3**: Once all flush messages are received, each process installs the new view and resumes normal operations.

“‘yaml

3. Assumptions for Correctness

1. **Reliable Communication**: The protocol assumes that messages between non-faulty processes are delivered reliably and in FIFO order.
2. **Synchronous Notifications**: The system assumes that processes are notified about view changes in a timely and consistent manner.
3. **Failure Model**: The protocol handles **crash failures** (where processes may stop functioning) but not Byzantine failures (where processes may act maliciously or unpredictably).
4. **Membership Stability**: During the execution of the view change protocol, it is assumed that the group membership does not change further until the new view is installed. The protocol can be extended to handle concurrent membership changes.

“‘yaml

4. Example Scenario

Imagine a distributed system where multiple servers replicate a data store and need to stay in sync: - If a server crashes, the remaining servers use virtual synchrony to ensure that all pending data updates are consistently applied before acknowledging the membership change. - New updates are only processed after the view change is completed, ensuring a consistent state among all active servers.

“‘yaml

Conclusion

Virtual synchrony ensures that all non-faulty members of a distributed system maintain a consistent view of messages and membership changes. The protocol relies on reliable communication, timely view change notifications, and crash failure handling to guarantee correctness. This approach is essential for applications requiring strong coordination and fault tolerance, such as distributed databases and replicated services. "'yaml