Department of Computer Engineering
Academic Term : January - May 2024

Class: B.E. (Computer) Semester VIII
Division: B Subject Name: Distributed Systems CSC 801

Experiment No 8

Roll No	9243
Name of Student	Pooja Jaya Banjan

Aim: Implement a simplified version of the MapReduce framework for distributed processing

Code:

```
def map(key, value, map_func):
  """
  Applies the map function to a key-value pair and emits intermediate pairs.

  Args:
     key: The input key.
     value: The input value.
     map_func: The map function to apply.

  Returns:
     A list of intermediate key-value pairs.
  """
  intermediate_pairs = []
  for key_out, value_out in map_func(key, value):
    intermediate_pairs.append((key_out, value_out))
  return intermediate_pairs

def reduce(key, values, reduce_func):
  """
  Applies the reduce function to a key and its associated values.

  Args:
```

```
    key: The intermediate key.
    values: A list of intermediate values for the key.
    reduce_func: The reduce function to apply.

  Returns:
    The final value for the key.
  """
  return reduce_func(key, values)

def mapreduce(data, map_func, reduce_func):
  """
  Executes the MapReduce job on a single machine.

  Args:
    data: An iterable containing key-value pairs.
    map_func: The map function to apply.
    reduce_func: The reduce function to apply.

  Returns:
    A dictionary containing the final results (key-value pairs).
  """
  # Apply map phase
  intermediate_data = {}
  for key, value in data:
    for key_out, value_out in map(key, value, map_func):
      if key_out not in intermediate_data:
        intermediate_data[key_out] = []
      intermediate_data[key_out].append(value_out)

  # Shuffle & sort (simulated)
  sorted_data = {}
  for key, values in intermediate_data.items():
    sorted_data[key] = sorted(values)

  # Apply reduce phase
  results = {}
  for key, values in sorted_data.items():
    results[key] = reduce(key, values, reduce_func)
```

```
    return results

# Example usage
def word_count_map(key, value):
    """
    Map function for word count example.
    """
    for word in value.split():
        yield (word, 1)

def word_count_reduce(key, values):
    """
    Reduce function for word count example.
    """
    return sum(values)

data = [("doc1", "This is a sample document"), ("doc2", "Another document with words")]
word_counts = mapreduce(data, word_count_map, word_count_reduce)

print(word_counts)
```

Output:

```
{'This': 1, 'is': 1, 'a': 1, 'sample': 1, 'document': 2, 'Another': 1, 'with': 1, 'words': 1}
```

Conclusion:

This simplified implementation of the MapReduce framework illustrates the foundational concepts of distributed data processing. By employing map and reduce functions, the framework efficiently manages input data, generating intermediate results that are organized and reduced to produce final outcomes. It effectively demonstrates the essence of parallel processing and the versatility of MapReduce in handling diverse data processing tasks. This introductory example, such as word counting, underscores the practical application of MapReduce in real-world scenarios. While this implementation offers a basic understanding, further exploration into distributed systems can unveil strategies for scaling MapReduce to address large-scale data processing requirements, enabling the development of scalable and efficient data processing solutions.

Post Lab Assignment:

1. Define Distributed System.

A distributed system is a collection of interconnected computers or devices that work together to achieve a common goal. In a distributed system, each computer, often referred to as a node or host, has its own processing power, memory, and resources, and they communicate and coordinate with each other through a network. The key characteristic of a distributed system is that it appears to its users as a single, unified system, even though it may consist of multiple independent components located across different physical locations.

Key features of distributed systems include:

1. Concurrency: Distributed systems allow multiple tasks or processes to run concurrently across different nodes, enabling parallel execution and improved performance.
2. Scalability: Distributed systems can scale horizontally by adding more nodes to the network, allowing them to handle increasing workloads and accommodate growing user demands.
3. Fault Tolerance: Distributed systems are designed to be resilient to failures, such as hardware failures, network outages, or software errors. They often employ redundancy, replication, and error recovery mechanisms to ensure continued operation in the event of failures.
4. Transparency: Distributed systems aim to provide transparency to users, hiding the complexities of the underlying network and infrastructure. This includes transparency of location, access, migration, and failure, making it appear as if resources are centralized and readily available.
5. Interoperability: Distributed systems support interoperability between different hardware and software platforms, enabling seamless communication and collaboration across heterogeneous environments.
6. Resource Sharing: Distributed systems allow for efficient resource sharing and utilization by enabling multiple users or applications to access and share resources such as processing power, storage, and data.

2. What is transparency?

   Distributed systems aim to provide transparency to users, hiding the complexities of the underlying network and infrastructure. This includes transparency of location, access, migration, concurrency and failure, making it appear as if resources are centralized and readily available.

1. Location Transparency: Users and applications are unaware of the physical location of resources (such as data or computing resources) in the distributed system. They can access resources using uniform interfaces without needing to know where the resources are located or how they are distributed across the network.
2. Access Transparency: Users and applications can access distributed resources using standard mechanisms and protocols, regardless of the underlying distribution and heterogeneity of the system. Access transparency ensures that users interact with distributed resources in a

consistent manner, regardless of differences in hardware, operating systems, or network protocols.

3. Concurrency Transparency: Distributed systems enable multiple tasks or processes to run concurrently across different nodes. Concurrency transparency ensures that users and applications are shielded from the complexities of concurrent execution, such as synchronization and communication between processes.
4. Failure Transparency: Distributed systems are designed to be resilient to failures, such as hardware failures or network partitions. Failure transparency ensures that users and applications experience minimal disruption in service in the event of failures. Failures are handled transparently by the system, and recovery mechanisms are invoked automatically to restore service.
5. Migration Transparency: Distributed systems may move resources or processes dynamically across different nodes in response to changing workload or system conditions. Migration transparency ensures that users and applications are unaware of resource migration and continue to interact with the system as if resources were stationary.