Question C

At Ormuco, we want to optimize every bits of software we write. Your goal is to write a new library that can be integrated to the Ormuco stack. Dealing with network issues everyday, latency is our biggest problem. Thus, your challenge is to write a new Geo Distributed LRU (Least Recently Used) cache with time expiration. This library will be used extensively by many of our services so it needs to meet the following criteria:

1 - Simplicity. Integration needs to be dead simple.

2 - Resilient to network failures or crashes.

3 - Near real time replication of data across Geolocation. Writes need to be in real time.

4 - Data consistency across regions

5 - Locality of reference, data should almost always be available from the closest region

6 - Flexible Schema

7 - Cache can expire

As a hint, we are not looking for quantity, but rather quality, maintainability, scalability, testability and a code that you can be proud of.

When submitting your code add the necessary documentation to explain your overall design and missing functionalities. Do it to the best of your knowledge.

ANSWER :-

Creating a distributed, geographically distributed cache with LRU (Least Recently Used) eviction, time-based expiration, and all the listed requirements is a complex task that often involves building a distributed system. You'll need to consider a combination of technologies and architectural choices to meet these criteria. Here's a high-level overview of how you might approach building such a system:

1. Simplicity:

Provide a straightforward API for integrating the cache into applications.

Implement a user-friendly configuration setup.

Use well-documented libraries and technologies.

2. Resilience to Network Failures or Crashes:

Use a distributed architecture with redundancy to ensure system availability.

Implement error handling and retry mechanisms for network issues.

Leverage load balancers and failover mechanisms for redundancy.

3. Near Real-Time Data Replication:

Use a distributed cache that supports real-time data replication across geolocations. Technologies like Redis, Apache Kafka, or distributed databases can be considered.

Implement publish-subscribe mechanisms or data streaming to keep data updated across regions in near real time.

4. Data Consistency Across Regions:

Use data synchronization mechanisms and techniques, such as quorum-based consistency models, to ensure data consistency across geolocations.

Leverage distributed databases with strong consistency guarantees if required.

5. Locality of Reference:

Implement a data routing mechanism to ensure that clients access data from the nearest geographic region. Content Delivery Networks (CDNs) and global load balancers can help with this.

6. Flexible Schema:

Use schema-less databases or NoSQL databases to allow for a flexible schema.

Consider document-oriented databases or key-value stores that can adapt to changing data structures.

7. Cache Expiry:

Implement time-based eviction policies (e.g., LRU with TTL) for cache entries.

Use in-memory data stores with automatic expiration policies.

Additional Considerations:

Load balancing and horizontal scaling: Implement load balancing to distribute client requests evenly and allow horizontal scaling for handling increased loads.

Monitoring and alerting: Set up monitoring and alerting for system health, performance, and failure detection.

Security: Implement encryption, authentication, and access control mechanisms to secure the cache and the data.

The specific technologies and architecture you choose will depend on your requirements, including the expected data volume, read/write patterns, and geographic distribution of your services. Popular technologies like Redis, Apache Kafka, and cloud-based solutions like AWS ElastiCache and Azure Cache for Redis are commonly used in building distributed caching systems. The choice of technologies should align with the requirements and the existing technology stack at Ormuco.

Building such a system is a substantial project and requires careful design and testing to ensure it meets the criteria and performs reliably in a geographically distributed environment. It may also require ongoing maintenance and optimization to achieve the desired performance and reliability.

Basic LRUChache mechanism code is given below:

```
public class CacheEntry {
        private Object key;
        private Object value;
        private long expirationTime;
        public CacheEntry(Object key, Object value, long expirationTime) {
                this.key = key;
                this.value = value;
                this.expirationTime = expirationTime;
        }
        public Object getKey() {
                return key;
        }
        public void setKey(Object key) {
                this.key = key;
        }
        public Object getValue() {
                return value;
        }
        public void setValue(Object value) {
                this.value = value;
        }
        public long getExpirationTime() {
                return expirationTime;
        }
        public void setExpirationTime(long expirationTime) {
                this.expirationTime = expirationTime;
        }
}
```

```java
import java.util.LinkedHashMap;
import java.util.Map;

public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private final int maxSize;

    public LRUCache(int maxSize) {
        super(maxSize, 0.75f, true);
        this.maxSize = maxSize;
    }

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > maxSize;
    }
}


import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
public class CacheManager {
    private final ConcurrentMap<String, LRUCache<Object, CacheEntry>> cacheMap = new ConcurrentHashMap<>();
    public void put(String region, Object key, Object value, long expirationTime) {
        CacheEntry entry = new CacheEntry(key, value, expirationTime);
        cacheMap.computeIfAbsent(region, r -> new LRUCache<>(1000)).put(key, entry);
        // You can handle synchronization depending on your environment
    }
    public Object get(String region, Object key) {
        CacheEntry entry = cacheMap.get(region).get(key);
        if (entry != null && System.currentTimeMillis() <= entry.getExpirationTime()) {
            return entry.getValue();
        } else {
            cacheMap.get(region).remove(key);
            return null;
        }
    }
}
```