**Question A**

Your goal for this question is to write a program that accepts two lines (x1,x2) and (x3,x4) on the x-axis and returns whether they overlap. As an example, (1,5) and (2,6) overlaps but not (1,5) and (6,8).

**Answer :-**

```
import java.util.Scanner;
public class LineSegmentOverlap {
	public static boolean doSegmentsOverlap(int x1, int x2, int x3, int x4) {
		// Here we are checking two line segments overlap by comparing their endpoint
		// values.
		// Segments (x1, x2) and (x3, x4) can be overlap if x2 is greater than or equal
		// to x3,
		// and x4 is greater than or equal to x1.
		return x2 >= x3 && x4 >= x1;
	}
	public static void main(String[] args) {
		// Input: Take two pairs of points (x1, x2) and (x3, x4) from the user.
		try (Scanner scanner = new Scanner(System.in)) {
			System.out.print("Enter the endpoints of the first line segment (x1 x2): ");
			int x1 = scanner.nextInt();
			int x2 = scanner.nextInt();

			System.out.print("Enter the endpoints of the second line segment (x3 x4): ");
			int x3 = scanner.nextInt();
			int x4 = scanner.nextInt();

			// Check if the line segments overlap and display the result.
			boolean overlap = doSegmentsOverlap(x1, x2, x3, x4);
			if (overlap) {
				System.out.println("The line segments overlap.");
			} else {
				System.out.println("The line segments do not overlap.");
			}
		}
	}
}
```

In this program, the doSegmentsOverlap method takes four integers representing the endpoints of two line segments, (x1, x2) and (x3, x4), and returns true if they overlap and false if they do not.

The main method provides sample input values and checks if the line segments overlap. This program will output whether the two line segments overlap or not based on the input values provided in the main method.

Using TDD approach, all test cases are defined in LineSegmentOverlapTest.java. (available in github)

**Question B**

The goal of this question is to write a software library that accepts 2 version string as input and returns whether one is greater than, equal, or less than the other. As an example: “1.2” is greater than “1.1”. Please provide all test cases you could think of.

**ANSWER**

Java software library that accepts two version strings as input and determines whether one is greater than, equal to, or less than the other:

```
public class VersionComparer {
	public static int compareVersions(String version1, String version2) {
		// Split version strings into arrays of string
		String[] parts1 = version1.split("\\.");
		String[] parts2 = version2.split("\\.");

		// Compare version components
		for (int i = 0; i < Math.max(parts1.length, parts2.length); i++) {
			int v1 = (i < parts1.length) ? Integer.parseInt(parts1[i]) : 0;
			int v2 = (i < parts2.length) ? Integer.parseInt(parts2[i]) : 0;
			if (v1 < v2) // version1 is less than version2
				return -1;
			if (v1 > v2) // version1 is greater than version2
				return 1;
		}
		return 0; // version1 is equal to version2
	}
	public static void main(String[] args) {
		String version1 = "1.2";
		String version2 = "1.1";
		int result = compareVersions(version1, version2);
		if (result < 0) {
			System.out.println('"' + version1 + '"' + " is less than " + '"' + version2 + '"' + ".");
		} else if (result > 0) {
			System.out.println('"' + version1 + '"' + " is greater than " + '"' + version2 + '"' + ".");
		} else {
			System.out.println('"' + version1 + '"' + " is equal to " + '"' + version2 + '"' + ".");
		}
	}
}
```

This program defines a VersionComparer class with a compareVersions method, which takes two version strings and compares them component by component. The result is -1 if the first version is less than the second, 1 if it's greater, and 0 if they are equal.

**Here are some test cases for the library:**

compareVersions("1.2", "1.1") should return "1.2 is greater than 1.1."

compareVersions ("1.1", "1.2") should return "1.1 is less than 1.2."

compareVersions ("1.2", "1.2") should return "1.2 is equal to 1.2."

compareVersions ("1.2.3", "1.2.3.4") should return "1.2.3 is less than 1.2.3.4."

compareVersions ("2.0", "1.9.9.9") should return "2.0 is greater than 1.9.9.9."

compareVersions ("1.0.0.0", "1.0") should return "1.0.0.0 is equal to 1.0."

compareVersions ("1.0", "1.0.0.0") should return "1.0 is equal to 1.0.0.0."

These test cases cover scenarios with varying numbers of components and different values, ensuring that the compareVersions method works correctly.

Using TDD approach, I have defined more test cases in below Junit test case file (also uploaded in github).

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
class VersionComparerTest {
	@Test
	void testCompareVersions() {
		assertEquals(1, VersionComparer.compareVersions("1.2", "1.1"));
		assertEquals(-1, VersionComparer.compareVersions("1.1", "1.2"));
		assertEquals(0, VersionComparer.compareVersions("1.2", "1.2"));
		assertEquals(1, VersionComparer.compareVersions("2.0.1", "2.0"));
		assertEquals(-1, VersionComparer.compareVersions("2.0", "2.0.1"));
		assertEquals(1, VersionComparer.compareVersions("1.2.3.4", "1.2.3.3"));
		assertEquals(-1, VersionComparer.compareVersions("1.2.3.3", "1.2.3.4"));
		assertEquals(0, VersionComparer.compareVersions("1.2.3.4", "1.2.3.4"));
		assertEquals(0, VersionComparer.compareVersions("1.0.0.0", "1.0"));
		assertEquals(0, VersionComparer.compareVersions("1.0", "1.0.0.0"));
		assertEquals(-1, VersionComparer.compareVersions("1.9.9.9", "2.0"));
	}
	@Test
	void testCompareVersionsWithDifferentLengths() {
		assertEquals(1, VersionComparer.compareVersions("1.2.3", "1.2"));
		assertEquals(-1, VersionComparer.compareVersions("1.2", "1.2.3"));
	}
	@Test
	void testCompareVersionsWithSingleDigit() {
		assertEquals(0, VersionComparer.compareVersions("1", "1"));
		assertEquals(1, VersionComparer.compareVersions("2", "1"));
		assertEquals(-1, VersionComparer.compareVersions("1", "2"));
	}
}
```

**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;
	}
}
```

```
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;
        }
    }
}
```