Name: **KIPRONO NAAMAN**
Reg no: **S13/02357/19**
Unit : **COMP 473 DISTRIBUTED SYSTEMS**
**CAT 2**

**Questions:**

**1)** Using the concepts of distributed systems learned in previous topics
and do own your research case study to
**a) Outline architecture of the original Google search engine**

The original Google search engine was built using a distributed system architecture that allowed for scalability and fault tolerance. The architecture consisted of the following components:

1. Web Crawler: The web crawler component was responsible for crawling the web and collecting pages for indexing. The crawler was distributed across multiple machines, and each machine would crawl a different subset of the web.
2. PageRank Algorithm: The Google search engine used the PageRank algorithm to rank the search results. PageRank is a link analysis algorithm that assigns a numerical weight to each page on the web based on the number and quality of links pointing to it. The pages with the highest PageRank scores were ranked higher in the search results.
3. Indexer: The indexer component was responsible for analyzing the web pages and building an index of keywords and their corresponding URLs. The indexer used a distributed hash table to store the index, and the hash table was sharded across multiple machines to allow for scalability.
4. Query Processor: The query processor component was responsible for processing user queries and retrieving relevant pages from the index. The query processor used a distributed lookup table to locate the shards containing the relevant index data, and then retrieved the data from those machines.
5. Query Expansion: The Google search engine used query expansion techniques to improve the relevance of search results. For example, if a user entered a search query for "cats", the search engine might also include related terms such as "kittens" and "felines" to broaden the search and provide more relevant results.

6. Load Balancer: The load balancer component was responsible for distributing user requests across the query processors. The load balancer used a distributed load balancing algorithm to ensure that the query processors were evenly utilized.
7. Caching: The Google search engine used caching to store frequently accessed web pages and reduce the load on the web servers. When a user searched for a query, the search engine would first check its cache to see if it already had a copy of the page. If the page was in the cache, it could be served directly from there without having to fetch it from the web server.
8. Storage System: The storage system component was responsible for storing and replicating the index data and other metadata across multiple machines to provide fault tolerance and ensure data availability.
9. AdWords: In addition to the search results, the Google search engine also displayed advertisements through its AdWords program. Advertisers could bid on keywords related to their products or services and have their ads displayed alongside the search results for those keywords. This generated significant revenue for Google and helped to fund the development of its search engine and other services.

**b) Google as a cloud provider**

In addition to being a search engine, Google is also a cloud provider with its Google Cloud Platform (GCP) offering various cloud services such as computing, storage, and networking to businesses and developers.

GCP provides a range of services that can be used for building and deploying distributed systems, including:

1. Google Compute Engine (GCE): This is Google's infrastructure as a service (IaaS) offering, which allows users to run virtual machines (VMs) in the cloud. It provides flexible and scalable compute resources, allowing users to create and manage VMs using a web interface, command-line tools, or API calls.
2. Google Kubernetes Engine (GKE): This is a managed Kubernetes service offered by Google, which provides a platform for deploying, managing, and scaling containerized applications in the cloud. GKE automates many tasks associated with managing Kubernetes clusters, such as scaling, upgrades, and monitoring.

3. Google Cloud Storage: This is Google's object storage service, which provides scalable and durable storage for unstructured data. It supports a variety of storage classes, ranging from standard to archival, and provides features such as lifecycle management, versioning, and access control.
4. Google Cloud Bigtable: This is a NoSQL database service offered by Google, which provides a scalable and high-performance datastore for large-scale applications. Bigtable is used by many Google products, such as Google Search and Google Maps, and supports features such as row-level transactions, automatic sharding, and high availability.
5. Google Cloud Pub/Sub: This is a messaging service offered by Google, which provides a reliable and scalable platform for asynchronous messaging between applications. It supports publish-subscribe and push-pull models, and provides features such as message ordering, filtering, and dead-letter queues.
6. Google Cloud AI Platform: This is a machine learning platform offered by Google, which provides a set of tools and services for building, training, and deploying machine learning models in the cloud. It supports popular machine learning frameworks such as TensorFlow and scikit-learn, and provides features such as hyperparameter tuning, distributed training, and model serving.
7. Cloud Dataflow: This is a fully managed service for developing and executing data processing pipelines. It enables users to create data pipelines that can ingest, transform, and analyze large volumes of data.

**c) Example Google Applications**

1. Google Search - allows users to search the internet for information using keywords and phrases.
2. Google Maps - provides mapping and location-based services, including satellite imagery, street maps, and route planning.
3. Google Drive - a cloud-based storage and collaboration platform that allows users to store, share and edit files and documents.
4. Gmail - a free email service that provides users with a secure and reliable way to send and receive emails.
5. Google Docs - a web-based word processing application that allows users to create and edit documents online.

6. YouTube - a video sharing platform that allows users to upload, share, and view videos.
7. Google Translate - a language translation service that allows users to translate text, speech, images, and web pages from one language to another.
8. Google Photos - a cloud-based photo storage and sharing service that allows users to backup, organize, and share their photos and videos.
9. Google Meet - a video conferencing platform that allows users to host and join virtual meetings, webinars, and events.
10. Google Assistant - an AI-powered virtual assistant that helps users with a range of tasks, including answering questions, setting reminders, and controlling smart home devices.
11. Firebase - a mobile and web application development platform that provides developers with tools and services to build and manage their apps.

**d) Google Physical model**

Google also has a physical infrastructure model that supports its cloud services and applications. The physical infrastructure is comprised of a global network of data centers that are interconnected through high-speed fiber optic cables. This physical infrastructure allows Google to provide reliable and fast access to its cloud services and applications from anywhere in the world.

Google's physical infrastructure includes multiple data centers located in various regions around the world. Each data center consists of thousands of servers, storage devices, and networking equipment that work together to provide the necessary computing resources to run Google's services and applications.

Google also invests heavily in renewable energy to power its data centers. The company has committed to purchasing enough renewable energy to match 100% of the energy used by its global operations. This includes investments in wind and solar power, as well as agreements to purchase energy from renewable sources.

Overall, Google's physical infrastructure is a critical component of its cloud services and applications, allowing the company to provide reliable and fast access to its services while also minimizing its environmental impact.

## e) Overall system architecture Key requirements

The overall system architecture key requirements for a distributed system can vary depending on the specific use case and application. However, there are some common requirements that apply in most cases.

1. Scalability: The system should be designed to scale horizontally or vertically as the demand for resources increases or decreases. This requires the ability to add or remove computing nodes dynamically, as needed.
2. Fault-tolerance: The system should be designed to handle failures and errors, such as hardware failures, network failures, or software errors. This can be achieved through redundancy, failover mechanisms, and error recovery.
3. Performance: The system should be designed to achieve high performance and low latency. This can be achieved through load balancing, caching, and optimization techniques.
4. Security: The system should be designed to protect against unauthorized access, data breaches, and other security threats. This can be achieved through authentication, encryption, and access control mechanisms.
5. Interoperability: The system should be designed to work with different operating systems, programming languages, and hardware platforms. This can be achieved through standardization, APIs, and other integration mechanisms.
6. Data management: The system should be designed to manage large amounts of data efficiently and reliably. This can be achieved through data partitioning, replication, and distribution techniques.
7. Monitoring and management: The system should be designed to provide visibility into its performance, health, and status. This can be achieved through monitoring tools, logging, and management consoles.

## f) Underlying communication paradigms

The underlying communication paradigms used in a distributed system depend on the specific system's requirements and design. However, there are some common communication paradigms that are frequently used in distributed systems:

1. Remote Procedure Call (RPC): RPC allows a process to execute a function or procedure in another process, possibly on a different machine, as if it were a local function call.

2. Message Passing: In this paradigm, processes communicate with each other by exchanging messages. A process sends a message to another process, and the receiving process responds appropriately.
3. Publish/Subscribe: In this paradigm, a publisher broadcasts messages to subscribers who have expressed an interest in receiving them.
4. Distributed Shared Memory (DSM): DSM allows processes in a distributed system to access a shared memory space as if it were local to each process. The DSM system is responsible for maintaining consistency between the different processes.
5. Data Streaming: Data streaming allows for the continuous flow of data from one process to another, typically using a push-based model. This paradigm is useful for real-time applications such as video streaming and financial data analysis.
6. Peer-to-Peer (P2P): In this paradigm, nodes in the system communicate directly with each other without a central server. This can provide better scalability and fault tolerance compared to client-server architectures.

These communication paradigms can be used individually or in combination to achieve the required functionality and performance for a specific distributed system.

**g) The Google File System as a data storage and Coordination Services**

The Google File System (GFS) is a distributed file system that was designed to provide reliable, scalable, and efficient storage for large-scale data processing applications. It is used as the primary storage system for many of Google's services, including Google Search, Google Analytics, and Google Maps.

One of the key requirements of the GFS architecture is its ability to provide fault tolerance and high availability. To achieve this, the system is designed to store multiple copies of each file on different servers. This ensures that if one server fails, the data can be retrieved from another copy.

Another key requirement is scalability. The GFS is designed to handle extremely large files, and can store petabytes of data across thousands of servers. To achieve this, the system is designed to distribute the load across multiple servers, and to optimize data access and retrieval for different types of applications.

The GFS uses a distributed coordination system to manage file operations and ensure consistency across the distributed file system. This system is based on a master/slave architecture, where a master node is responsible for managing the metadata for the file system, and multiple slave nodes are responsible for storing and retrieving data.

The underlying communication paradigms used in the GFS are based on a combination of RPC (Remote Procedure Calls) and TCP/IP protocols. RPC is used to facilitate communication between the master and slave nodes, while TCP/IP is used for data transmission and retrieval.

Overall, the GFS is a complex system that requires a robust architecture, fault tolerance, and scalability to meet the demands of large-scale data processing applications. The use of distributed coordination and communication protocols is critical to the system's success, as it allows for efficient data storage and retrieval across thousands of servers.

**h) Overall architecture of Bigtable**

Bigtable is a distributed storage system designed by Google to handle large amounts of structured data. It is a NoSQL database that allows users to store and retrieve data using key-value pairs. Here is an overview of its overall architecture:

1. Clients: Applications interact with Bigtable through client libraries that provide APIs for data storage and retrieval.
2. Master node: The master node coordinates the operations of the entire system. It is responsible for assigning tablets to tablet servers, monitoring their health, and directing clients to the appropriate tablet server.
3. Tablet servers: Tablet servers are responsible for storing and serving data. Each tablet server is assigned a set of tablets, and each tablet contains a subset of the data stored in Bigtable.
4. Tablets: Tablets are the basic unit of data storage in Bigtable. They are assigned to tablet servers by the master node, and each tablet is responsible for a range of rows in a table.
5. Chubby: Chubby is a distributed lock service that provides synchronization and coordination among the different components of Bigtable. It is used to elect a master node and to ensure that only one master node is active at any given time.

Overall, the architecture of Bigtable is designed for scalability, fault-tolerance, and high performance. By using tablets to partition data and distributing them across multiple tablet servers, Bigtable can handle large amounts of data and provide fast access to it. The use of Chubby for coordination ensures that the system is highly available and reliable, even in the face of failures.

**i) The overall execution of a MapReduce program**

MapReduce is a programming model and a processing framework designed for processing large datasets in parallel across a distributed cluster of computers. It has two primary operations: Map and Reduce.

The overall execution of a MapReduce program can be broken down into the following steps:

1. Input Splitting: The input data is divided into smaller pieces or chunks, called input splits. The size of these splits can be configured based on the size of the input data and the processing capacity of the cluster.
2. Mapping: The input splits are processed by a set of map tasks in parallel. The input key-value pairs are transformed into intermediate key-value pairs using the user-defined map function.
3. Shuffling and Sorting: The intermediate key-value pairs are partitioned and sorted based on their keys. The shuffle and sort phase redistributes the data across the cluster based on the keys, ensuring that all the values associated with a given key are sent to the same reduce task.
4. Reducing: The intermediate key-value pairs are processed by a set of reduce tasks in parallel. The reduce task takes the intermediate key-value pairs and produces the final output key-value pairs using the user-defined reduce function.
5. Output: The final output key-value pairs are written to the output storage system, such as a distributed file system or a database.

The MapReduce programming model and processing framework are widely used for large-scale data processing tasks, such as batch processing, log processing, and data mining. It provides fault tolerance, scalability, and parallelism, making it a powerful tool for processing big data.

**j) Google security policies and security mechanism**

Google has a variety of security policies and mechanisms to protect its systems and user data. Here are some of the key policies and mechanisms:

1. Multi-layered security: Google has a multi-layered security architecture that includes physical security, network security, operating system security, application security, and data security.

2. Encryption: Google encrypts all data in transit and at rest. Google also offers encryption for data at rest in its cloud storage services, and encrypts data using industry-standard algorithms such as AES256.
3. Authentication and access control: Google uses two-factor authentication and access controls to ensure that only authorized personnel can access its systems and data. Google also employs strict access controls for its APIs, web interfaces, and management consoles.
4. Auditing and monitoring: Google has a comprehensive audit and monitoring system in place to detect and prevent security breaches. It uses machine learning algorithms to detect anomalous activity and performs regular security assessments and penetration testing.
5. Disaster recovery and business continuity: Google has a robust disaster recovery and business continuity plan in place, which includes data backups, replication, and failover mechanisms.
6. Bug bounty program: Google has a bug bounty program that rewards individuals who report security vulnerabilities in its systems and software.
7. Compliance: Google is compliant with several industry standards and regulations, such as ISO 27001, SOC 2, and GDPR. Google also provides compliance certifications for its cloud services to help customers meet their own regulatory requirements.

Overall, Google's security policies and mechanisms are designed to protect its systems and user data from a wide range of threats, including cyberattacks, data breaches, and unauthorized access.

**2) Using same handlines in assignment (1) summarize and make your own brief notes on Amazon platform as distributed systems application.**

Amazon is a leading provider of cloud computing services and offers a wide range of distributed systems applications. These applications are designed to help businesses and individuals store, process, and manage large amounts of data, as well as provide various other computing services. Some of the key services offered by Amazon include:

a) Amazon Elastic Compute Cloud (EC2): A web service that provides scalable computing capacity in the cloud. This allows businesses and individuals to deploy virtual machines and run applications on Amazon's infrastructure.

b) Amazon Simple Storage Service (S3): A scalable, high-speed, low-cost, web-based cloud storage service designed for online backup and archiving of data and applications.

This service enables businesses to store and retrieve any amount of data from anywhere on the web.

c) Amazon Relational Database Service (RDS): A web service that makes it easy to set up, operate, and scale a relational database in the cloud. This service provides businesses with the flexibility to choose their preferred database engine and the ability to scale their database infrastructure as needed.

d) Amazon DynamoDB: A fast, flexible, and fully managed NoSQL database service designed for applications that require consistent, single-digit millisecond latency at any scale. This service allows businesses to store and retrieve any amount of data, while providing built-in security, backup, and disaster recovery capabilities.

e) Amazon Simple Queue Service (SQS): A fully managed message queuing service that enables businesses to decouple and scale microservices, distributed systems, and serverless applications.

f) Amazon Simple Notification Service (SNS): A fully managed pub/sub messaging service that enables businesses to send push notifications to mobile devices, distribute messages to serverless applications, and trigger workflows in other AWS services.

g) Amazon Elastic Container Service (ECS): A fully managed container orchestration service that enables businesses to run, scale, and manage Docker containers on Amazon's infrastructure.

h) Amazon Elastic Kubernetes Service (EKS): A fully managed Kubernetes service that makes it easy to deploy, manage, and scale containerized applications using Kubernetes on Amazon's infrastructure.

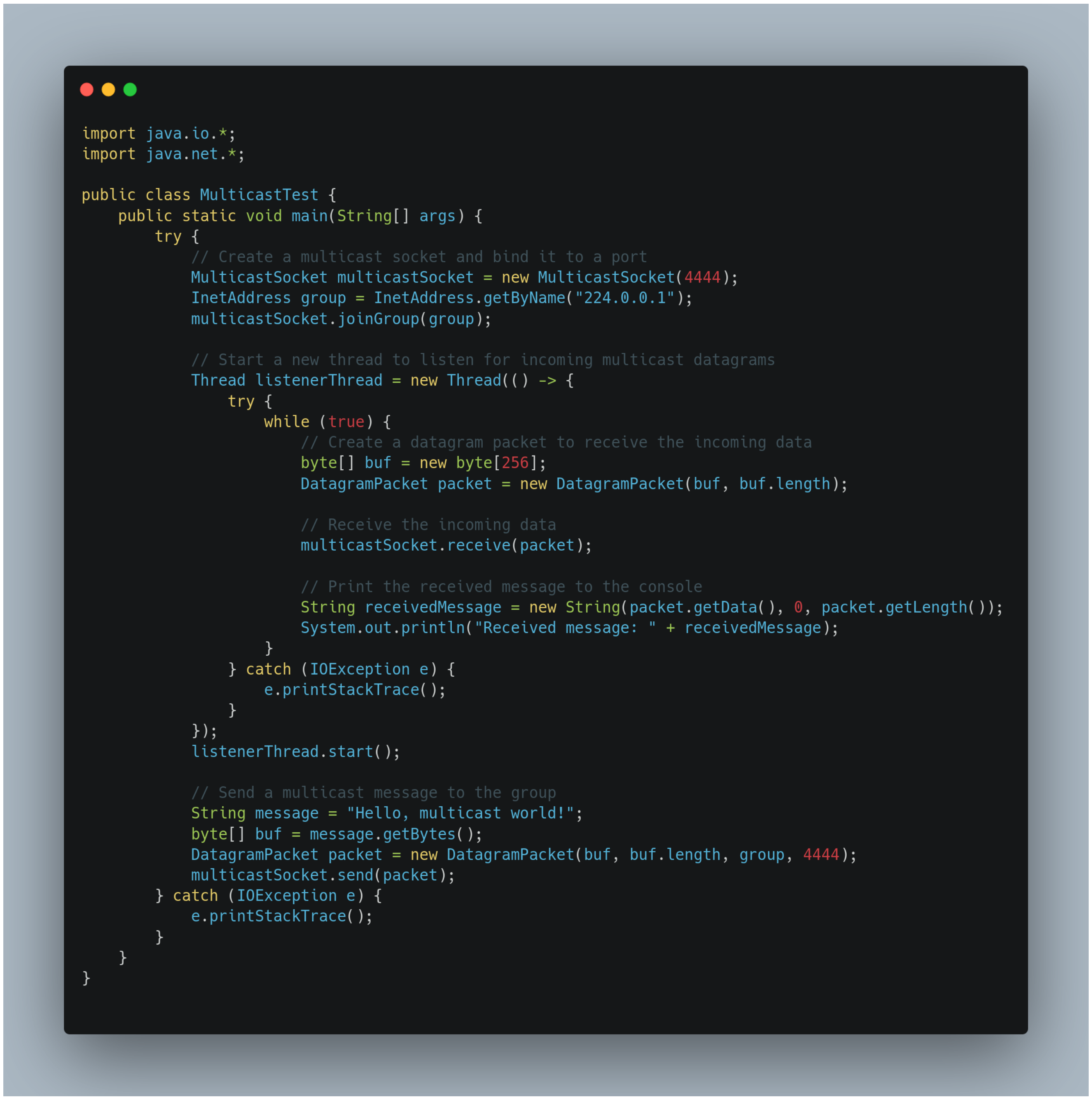
i) Amazon Lambda: A compute service that runs code in response to events and automatically manages the computing resources required by that code. This service allows businesses to build applications that respond quickly to new information or changes in data.

j) Amazon Web Services Identity and Access Management (IAM): A web service that helps businesses securely control access to AWS resources. This service allows businesses to grant specific permissions to users, groups, and roles, while ensuring that only authorized users can access their resources.

In addition to these services, Amazon also offers various other distributed systems applications, including Amazon Redshift, Amazon Kinesis, Amazon Elastic File System (EFS), Amazon API Gateway, and Amazon CloudFront. The underlying communication paradigms used in these applications include REST APIs, messaging queues, and event-driven architectures. Amazon's security policies and security mechanisms include multi-factor authentication, encryption, access control, and continuous monitoring. Overall, Amazon's platform is a powerful and comprehensive distributed systems application that can help businesses of all sizes manage their computing and data storage needs.

**2.)** Practical Case study :
Repeat the same concepts as in UDP and TCP communication examples given
above for the following middleware protocols.
**a) Java API to IP multicast communication i.e give running sample code for Multicast peer joins a group and sends and receives datagrams**
To test this code, we can run multiple instances of it on different machines and have them join the same multicast group. When one instance sends a datagram to the group, all instances in the group will receive it and send a reply back to the sender.
Below code creates a MulticastSocket and joins a multicast group specified by the IP address 224.0.0.1 and port number 4446. It then enters a loop where it waits to receive datagrams from the group using socket.receive(packet). Once it receives a datagram, it prints out the contents of the message and sends a reply to the sender using

socket.send(replyPacket).



```java
import java.io.*;
import java.net.*;

public class MulticastTest {
    public static void main(String[] args) {
        try {
            // Create a multicast socket and bind it to a port
            MulticastSocket multicastSocket = new MulticastSocket(4444);
            InetAddress group = InetAddress.getByName("224.0.0.1");
            multicastSocket.joinGroup(group);

            // Start a new thread to listen for incoming multicast datagrams
            Thread listenerThread = new Thread(() -> {
                try {
                    while (true) {
                        // Create a datagram packet to receive the incoming data
                        byte[] buf = new byte[256];
                        DatagramPacket packet = new DatagramPacket(buf, buf.length);

                        // Receive the incoming data
                        multicastSocket.receive(packet);

                        // Print the received message to the console
                        String receivedMessage = new String(packet.getData(), 0, packet.getLength());
                        System.out.println("Received message: " + receivedMessage);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            });
            listenerThread.start();

            // Send a multicast message to the group
            String message = "Hello, multicast world!";
            byte[] buf = message.getBytes();
            DatagramPacket packet = new DatagramPacket(buf, buf.length, group, 4444);
            multicastSocket.send(packet);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

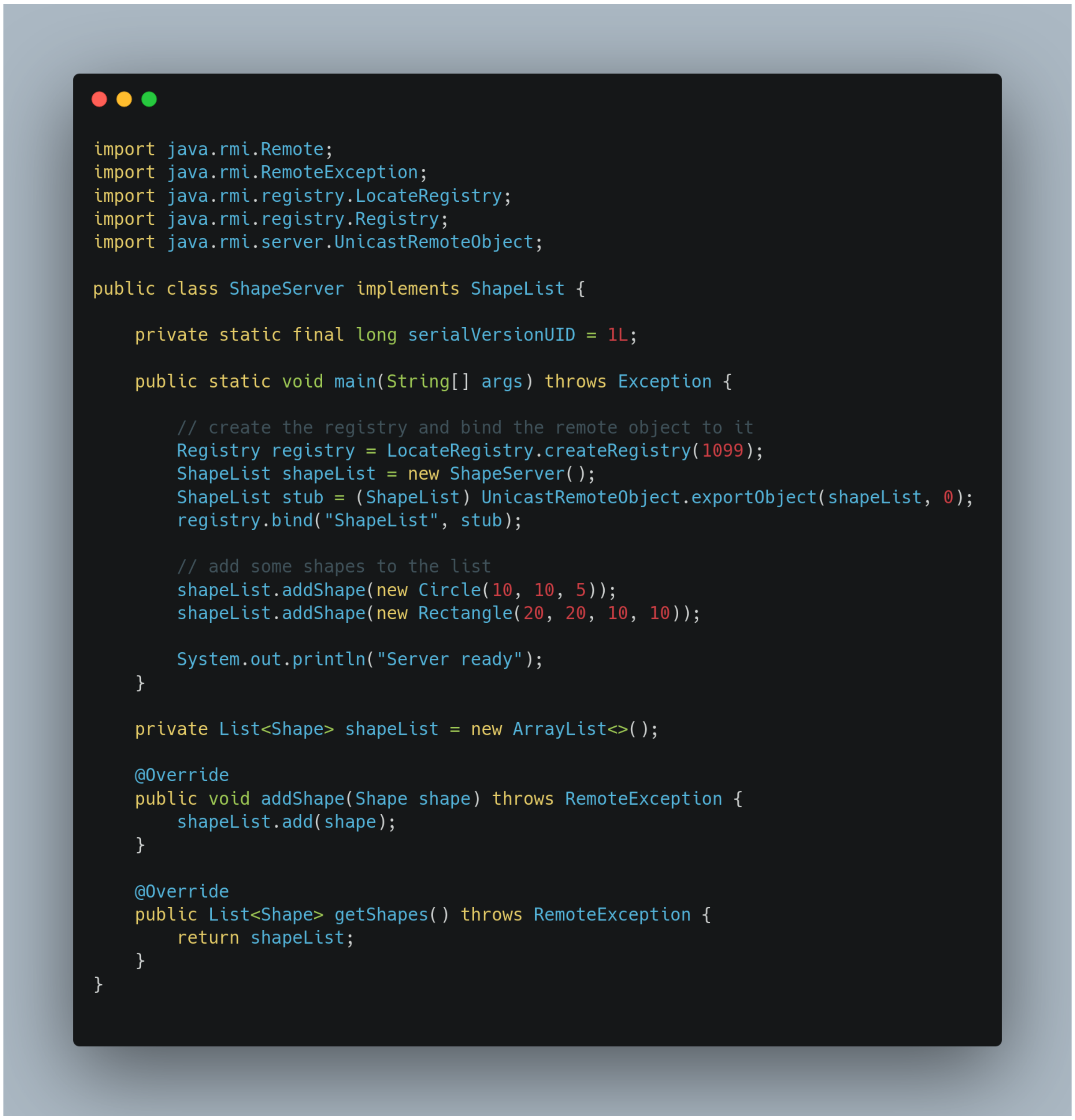
**b) Remote interfaces in Java RMI i.e give a running sample code for Java Remote interfaces Shape and Shape List on client side as well as server side.**

To demonstrates how to use Remote Method Invocation (RMI) to implement remote interfaces in Java. In this example, we have a Shape interface which defines methods for moving and getting the type of a shape, and a ShapeList interface which defines methods for adding shapes to a list and getting the list of shapes. The ShapeServer class implements the ShapeList interface and creates a registry for the remote object. The ShapeClient class looks up the remote object in the registry and uses it to add a new shape to the list and get the list of shapes.

The Shape and ShapeList interfaces extend the Serializable and Remote interfaces, respectively, to indicate that they can be serialized and used remotely. The ShapeServer class creates a registry and binds the remote object to it using the UnicastRemoteObject.exportObject() method. The ShapeServer class also adds some shapes to the list when it starts up.

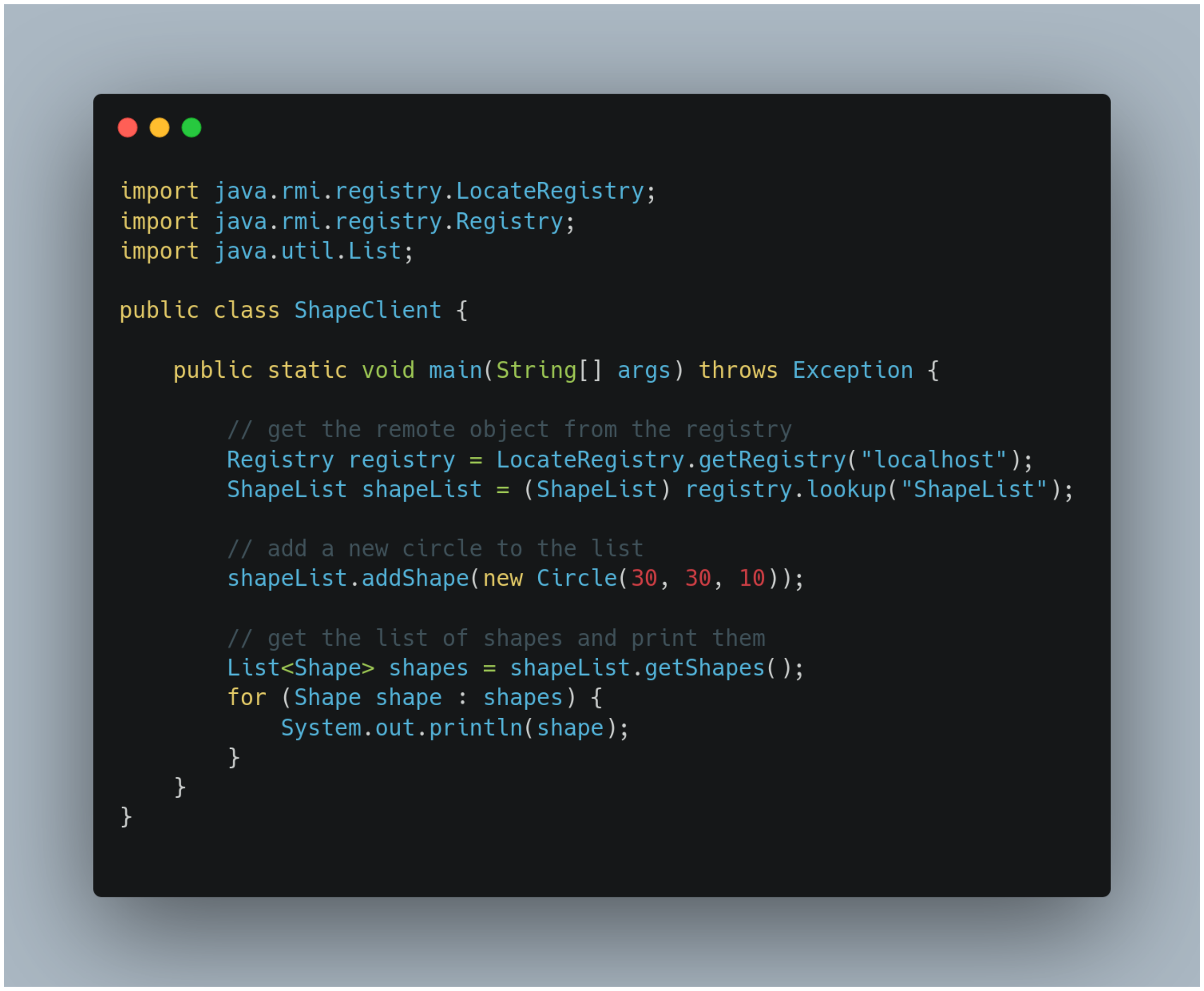
The ShapeClient class looks up the remote object in the registry using the LocateRegistry.getRegistry() method and the registry.lookup() method. The ShapeClient class then uses the

**Server side code;**



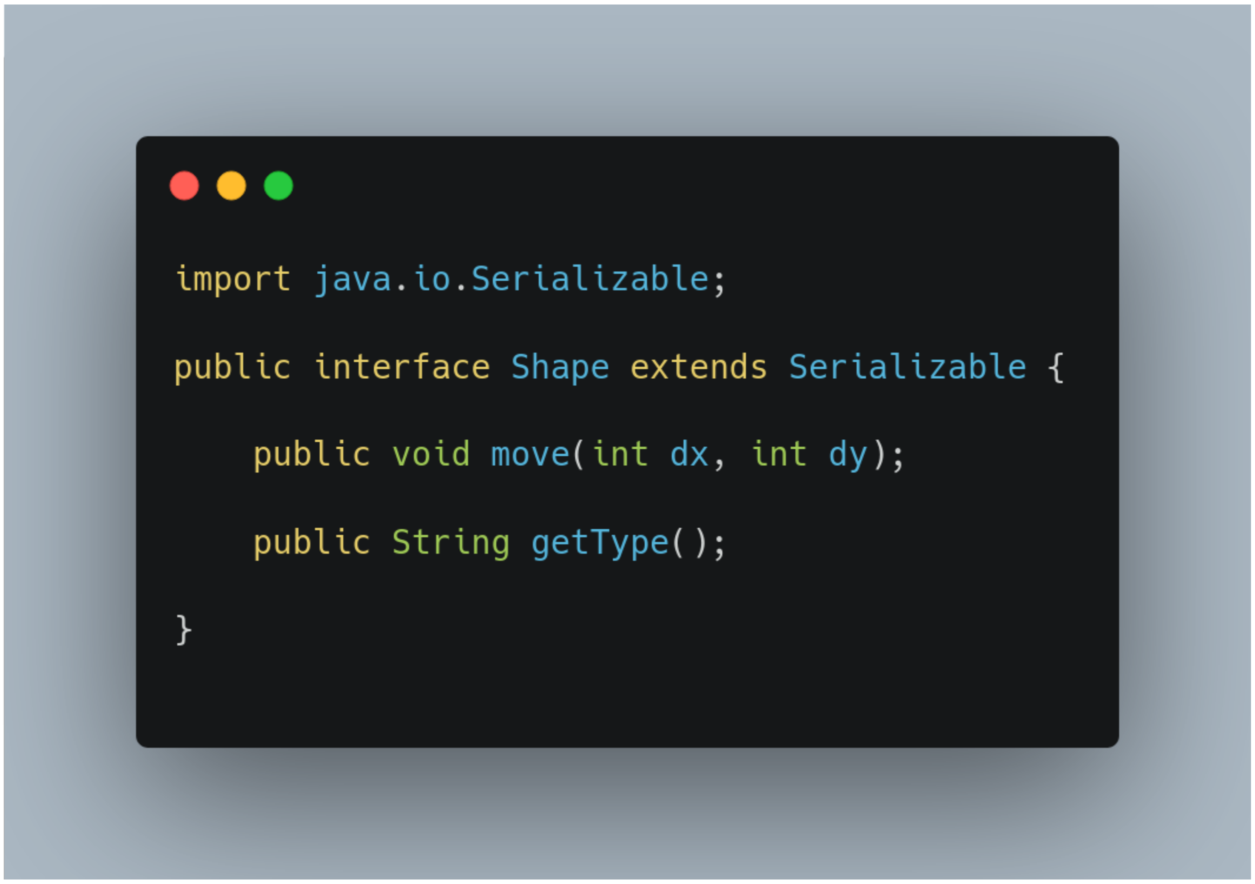
```java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class ShapeServer implements ShapeList {

    private static final long serialVersionUID = 1L;

    public static void main(String[] args) throws Exception {

        // create the registry and bind the remote object to it
        Registry registry = LocateRegistry.createRegistry(1099);
        ShapeList shapeList = new ShapeServer();
        ShapeList stub = (ShapeList) UnicastRemoteObject.exportObject(shapeList, 0);
        registry.bind("ShapeList", stub);

        // add some shapes to the list
        shapeList.addShape(new Circle(10, 10, 5));
        shapeList.addShape(new Rectangle(20, 20, 10, 10));

        System.out.println("Server ready");
    }

    private List<Shape> shapeList = new ArrayList<>();

    @Override
    public void addShape(Shape shape) throws RemoteException {
        shapeList.add(shape);
    }

    @Override
    public List<Shape> getShapes() throws RemoteException {
        return shapeList;
    }
}
```

**Client Side;**



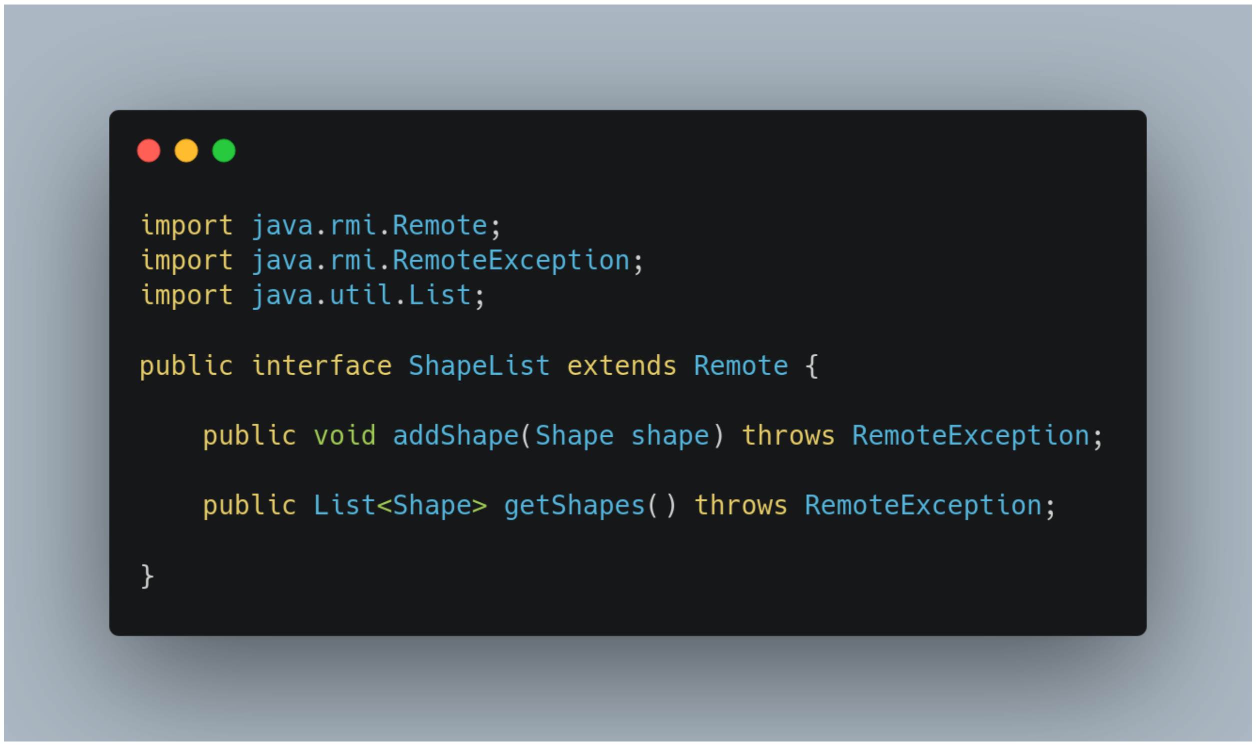
```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.List;

public class ShapeClient {

    public static void main(String[] args) throws Exception {

        // get the remote object from the registry
        Registry registry = LocateRegistry.getRegistry("localhost");
        ShapeList shapeList = (ShapeList) registry.lookup("ShapeList");

        // add a new circle to the list
        shapeList.addShape(new Circle(30, 30, 10));

        // get the list of shapes and print them
        List<Shape> shapes = shapeList.getShapes();
        for (Shape shape : shapes) {
            System.out.println(shape);
        }
    }
}
```

**Shape Interface**



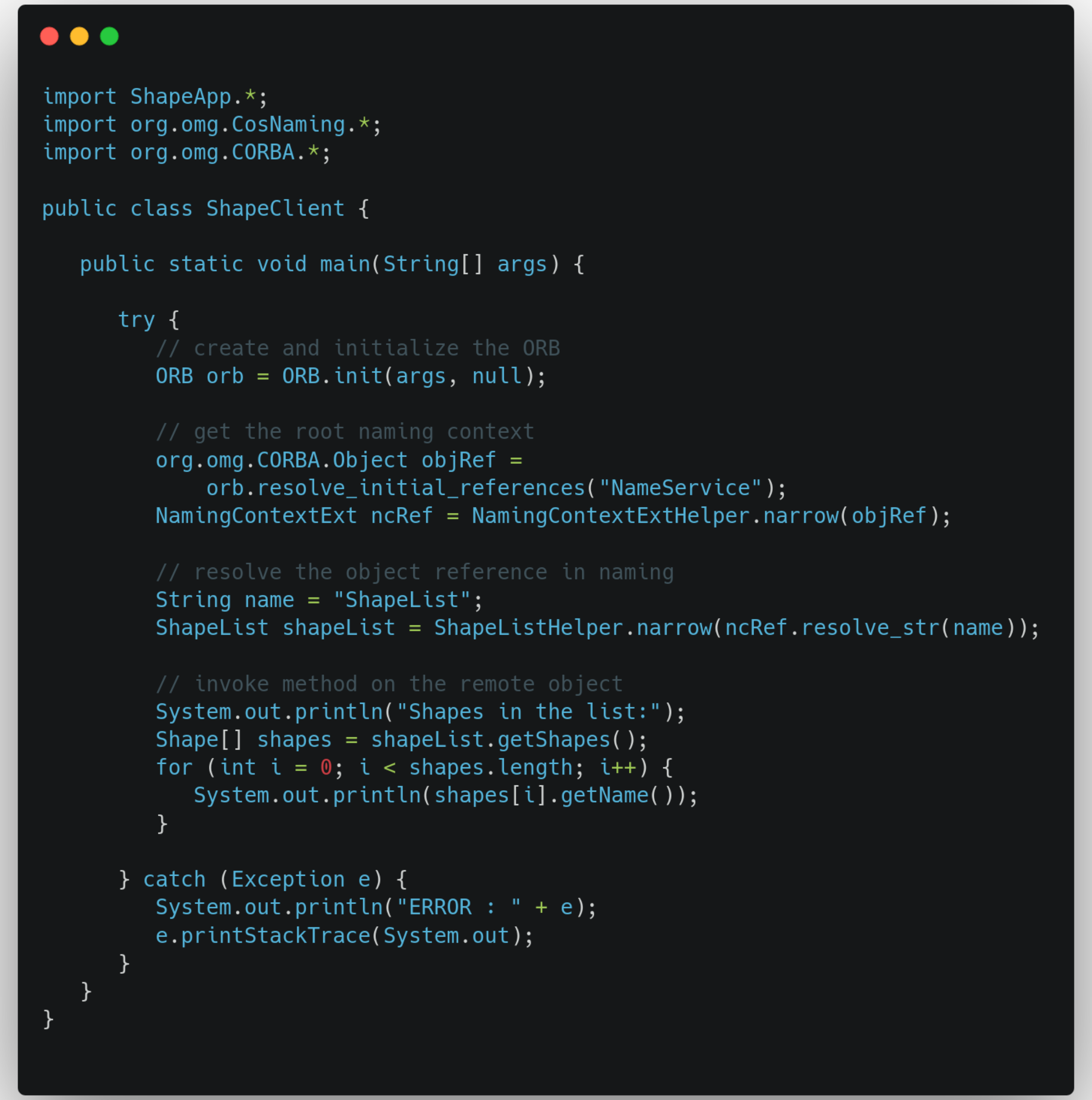
```java
import java.io.Serializable;

public interface Shape extends Serializable {

    public void move(int dx, int dy);

    public String getType();

}
```

**Shape List Interface**



```java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface ShapeList extends Remote {

    public void addShape(Shape shape) throws RemoteException;

    public List<Shape> getShapes() throws RemoteException;

}
```

**c) CORBA client and server i.e give a running sample code for Java client program for CORBA interfaces Shape and ShapeList**



```java
import ShapeApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ShapeClient {

   public static void main(String[] args) {

      try {
         // create and initialize the ORB
         ORB orb = ORB.init(args, null);

         // get the root naming context
         org.omg.CORBA.Object objRef =
             orb.resolve_initial_references("NameService");
         NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

         // resolve the object reference in naming
         String name = "ShapeList";
         ShapeList shapeList = ShapeListHelper.narrow(ncRef.resolve_str(name));

         // invoke method on the remote object
         System.out.println("Shapes in the list:");
         Shape[] shapes = shapeList.getShapes();
         for (int i = 0; i < shapes.length; i++) {
            System.out.println(shapes[i].getName());
         }

      } catch (Exception e) {
         System.out.println("ERROR : " + e);
         e.printStackTrace(System.out);
      }
   }
}
```