Remote Procedure Call
Remote Method Invocation.
Technologies Examples

Examples of RPC/RMI Technologies

- Procedural: expose **procedures/functions** over a network
  - Sun RPC (1984)
  - Microsoft RPC

- Distributed objects: allow methods of **remote objects** to be invoked as if they were local objects, hiding network details
  - CORBA (1991)
  - Java RMI (1997)
  - WCF (Windows Communication Foundation) (2006)
  - RPyC (2005)

- Service oriented: expose **service interfaces** with explicit contracts and messaging
  - gRPC (2015)

Examples of RPC/RMI Technologies.

Case Study1: Java RMI

https://docs.oracle.com/javase/tutorial/rmi/index.html

https://docs.oracle.com/en/java/javase/24/docs/api/java.rmi/java/rmi/package-summary.html

Java RMI

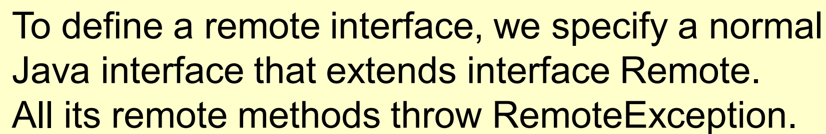
- **Java RMI** (Remote Method Invocation): RPC in Java
- Extends RPC to distributed objects
- Allows to develop distributed applications in Java where an object on one JVM invokes methods on an object in another JVM
- Goal: access transparency, but distribution transparency is still not full
- Obsolete now, but is discussed here as historical relevant concepts and as background for modern technologies like gRPC
- *Remote interface*: specifies set of methods to be invoked remotely
- *Remote object*: instance of a class that implements a remote interface
- *Remote method invocation*: invoke methods of a remote interface on a remote object
  - Goal: keep same syntax as local invocation

Steps to develop Java RMI application

1. Write Remote Interface. Since everything is Java, no need for an IDL (Interface is also Java)
2. Write implementation of remote interface (the class of remote objects)
3. Write a server program to instantiate remote objects.
4. Write a client
5. Generate stub and skeleton. 2 possibilities:
   A. Static generation: use rmic (JavaRMI IDL compiler) to produce the code of stub and skeleton classes
   B. Dynamic generation: stub and skeleton are generated transparently by the use of *reflection* in Java

Remote Interface: MathService Example

```
public interface MathService extends Remote {
    int add(int a, int b) throws RemoteException;
    int mult(int a, int b) throws RemoteException;
}
```



Remote Object Implementation: MathImpl

```
public class MathServiceImpl extends UnicastRemoteObject
                            implements MathService {

    protected MathServiceImpl() throws RemoteException {
        super();
    }

    @Override
    public int add(int a, int b) throws RemoteException {
        return a + b;
    }

    @Override
    public int mult(int a, int b) throws RemoteException {
        return a * b;
    }
}
```

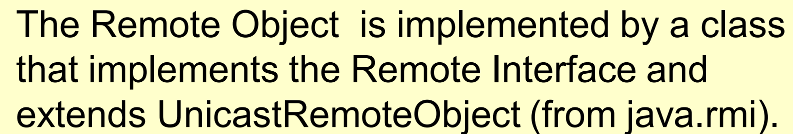
Remote Object Implementation: MathImpl

```
public class MathServiceImpl extends UnicastRemoteObject
                              implements MathService {

    protected MathServiceImpl() throws RemoteException {
        super();
    }

    @Override
    public int add(int
        return a +
    }

    @Override
    public int mult(int a,                                {
        return a * b;
    }
}
```



Server Program: MathServer

```
public class MathServer {
    public static void main(String[] args) {
      try {
        MathService mathService = new MathServiceImpl();
        Registry registry;
        try {
           // Try to locate an existing registry
            registry = LocateRegistry.getRegistry(1099);
            registry.list(); // test if registry is alive
           } catch (RemoteException e) {
                // Registry not found, create a new one
                registry = LocateRegistry.createRegistry(1099);
                }

            // Bind the service to the registry
            registry.rebind("MathService", mathService);
            System.out.println("MathService is running");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Server Program: MathServer

```
public class MathServer {
    public static void main(String[] args) {
      try {
        MathService mathService = new MathServiceImpl();
        Registry registry;
        try {
           // Try to locate an existing registry
            registry = LocateRegistry.getRegistry(1099);
            registry.list(); // test if registry is alive
           } catch (RemoteException e) {
                // Registry not found, create a new one
                registry = LocateRegistry.createRegistry(1099);
                }

            // Bind the service to t
            registry.rebind("MathSe
            System.out.println(
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The Server program instantiates a Remote Object and binds it in the registry to a name. The clients will look up the Remote Object by this name. The registry will handle incoming calls for this object and dispatch them to the actual remote object.

Client Program: MathClient

```
public class MathClient {
    public static void main(String[] args) {
        try {
            // Connect to RMI registry on localhost
            Registry registry =
                    LocateRegistry.getRegistry("localhost", 1099);

            // Look up the remote service
            MathService mathService = (MathService)
                                            registry.lookup("MathService");

            // Call remote methods
            int sum = mathService.add(5, 3);
            int product = mathService.mult(5, 3);

            System.out.println("5 + 3 = " + sum);
            System.out.println("5 * 3 = " + product);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

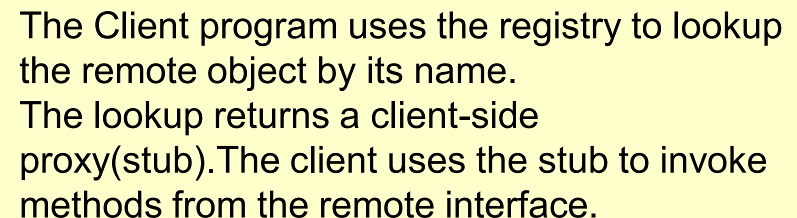
Client Program: MathClient

```
public class MathClient {
    public static void main(String[] args) {
      try {
         // Connect to RMI registry on localhost
         Registry registry =
                LocateRegistry.getRegistry("localhost", 1099);

         // Look up the remote service
         MathService mathService = (MathService)
                                    registry.lookup("MathService");

         // Call remote methods
         int sum = mathService.add(5,    ;
         int product = mathService.mult

         System.out.println("5 + 3 =
         System.out.println("5 *
        } catch (Exception e) {
           e.printStackTrace();
        }
    }
}
```



Stub generation

A. **Static generation:** use **rmic** (JavaRMI IDL compiler) to produce the code of stub and skeleton classes:

>**rmic** –v1.1 –keep MathServiceImpl

Arguments:

-v1.1 because after Java 1.2 the default is not to generate skeleton, if we want generated skeletons we have to compile with –v1.1

-keep to keep the source code of the generated classes (for the user to be able to inspect it)

Generated classes for stub and skeleton, using naming convention:

```
MathServiceImpl_Stub.class
MathServiceImpl_Skel.class
```

B. **Dynamic generation**: stub and skeleton are generated transparently by the use of ***reflection*** in Java

– Just compile the written code and ready to run serve and client!

Java RMI: other features

- Call semantics:
  - At-most-once(Default semantics):
    - A remote method is executed either once or not at all
    - Java RMI guarantees: No duplicate executions (built-in protection)
  - Maybe semantics:
    - If something goes wrong (network crash, timeout) client gets a **RemoteException** , execution state is **unknown** (because the call *may or may not* have executed)
  - Exactly once: impossible to guarantee
- Synchronous blocking client
- Concurrency: a remote method can be invoked concurrently by multiple clients
  - a remote object implementation needs to make sure its implementation is thread-safe
- Dynamic class loading: since class definitions are required for serializing/deserializing objects passed as parameters, RMI also provides a facility for dynamically loading class definitions

Examples of RPC/RMI Technologies.
Case Study2: gRPC

https://grpc.io/

gRPC

- Google's RPC platform initially, now open source available to all developers
  - Modern, high-performance framework
  - designed for cloud apps
  - Used by many companies and in many distributed systems; e.g. Google, Dropbox, Netflix, Square, etcd, CockroachDB
- Works across different OS, hardware and languages
  - Supports python, java, C++,C#, Go, Swift, Node.js, ….
- Uses http/2 as communication protocol
- Uses ProtoBuf both as an IDL and as a representation for serializing structured messages

gRPC: HTTP2

- Communicatin over HTTP/2
- Basic idea of gRPC: treat RPCs as references to HTTP objects
- HTTP/2: major revision of HTTP that provides significant performance benefits over HTTP 1.x
- HTTP/2 in a nutshell
  - Binary framing layer: HTTP/2 request/response is divided into small messages and framed in binary format, making message transmission efficient
  - From request/response messages to streams
    - Stream: bidirectional flow of bytes within an established connection, which may carry one or more messages
    - Message: complete sequence of frames that map to a logical request or response message
    - Frame: smallest unit of communication in HTTP/2, each containing a frame header, which at least identifies the stream to which the frame belongs
  - Request/response multiplexing (usage of a single connection per client): allows for efficient use of TCP connections and avoids head-of-line blocking at HTTP level
  - Native support for bidirectional streaming
  - HTTP header compression: to reduce protocol overhead

gRPC: Protobuff

- gRPC uses protocol buffers (protobufs) as:
  1. IDL to define service interface: automatic generation of client stubs and abstract server classes
  2. Message interchange format: gRPC messages are serialized using protocol buffers, thus resulting in small message payloads

- Protocol buffers:
  - An open-source mechanism (by Google) to serialize structured data
  - Binary data representation
  - Strongly typed
  - Data types are structured as messages
    - message: small logical record of information containing a series of name-value pairs called fields
    - Fields have unique field numbers (e.g., string name = 1), used to identify fields in message binary format

Steps to develop gRPC application

1. Write Service Interface. Use protobuf as an IDL
2. Generate code with the **protoc** compiler taking the IDL as input. Various targe languages are supported.
3. Write implementation of remote interface (the class of remote objects). It must extend the asdtract base class generated by protoc
4. Write a server program to host the service implementation.
5. Write a client

gRPC: Remote Interface: math.proto

```
syntax = "proto3";

package grpc1;

option java_multiple_files = true;

service MathService {
  rpc Add (BinaryOpRequest) returns (BinaryOpResponse);
  rpc Mult (BinaryOpRequest) returns (BinaryOpResponse);
}

message BinaryOpRequest {
  int32 a = 1;
  int32 b = 2;
}

message BinaryOpResponse {
  int32 result = 1;
}
```

gRPC: Remote Interface: math.proto

```
syntax = "proto3";

package grpc1;

option java_multiple_files = true;

service MathService {
  rpc Add (BinaryOpRequest) returns (BinaryOpResponse);
  rpc Mult (   aryOpRequest) returns (BinaryOpResponse);
}

message
```

To define a service interface, we specify a named service in the .proto file.
Then we define rpc methods inside our service definition, specifying their request and response types. gRPC lets you define four kinds of service methods. The kind of *simple RPC* is where the client sends a request to the server using the stub and waits for a response to come back, just like a normal function call.

Generate code from math.proto

- `protoc --java_out=. --grpc-java_out=. math.proto`

- The option --java_out generates java classes of interface definition
  - this generates: BinaryOpRequest, BinaryOpResponse,
- The option --grpc-java_out generates the service class in java: MathServiceGrpc.java
- This file contains:
  - MathServiceImplBase the abstract base class for server object (your MathImpl will have to extend it)
  - Client stubs: MathServiceBlockingStub if synchronous client, MathServiceStub if async client

gRPC: Remote Object Implementation: MathImpl

```
public class MathImpl extends MathServiceGrpc.MathServiceImplBase {

     @Override
     public void add(BinaryOpRequest request,
                StreamObserver<BinaryOpResponse> responseObserver) {

        int result = request.getA() + request.getB();
        BinaryOpResponse response = BinaryOpResponse.newBuilder()
                .setResult(result)
                .build();
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }

   @Override
    public void mult(BinaryOpRequest request,
                StreamObserver<BinaryOpResponse> responseObserver) {

        …
     }
}
```

gRPC: Remote Object Implementation: MathImpl

```
public class MathImpl extends MathServiceGrpc.MathServiceImplBase {

    @Override
    public void add(BinaryOpRequest request,
                StreamObserver<BinaryOpResponse>

        int re    t = request.getA() + request.getB(),
        Binar    sponse response = BinaryOpResponse.newBuilder()
                setResult(result)
                bu
                                nse);

                                t,
                                onse> responseObserver) {
```

Our serverobject must extend the generated abstract ImplBase class

A RPC service method takes two parameters:
BinaryOpRequest: the request, and
StreamObserver<BinaryOpResponse>: a response observer.
We construct and populate a BinaryOpResponse object to return to the client. We use the response observer's onNext() method to return the response. We use the response observer's onCompleted()
method to specify that we've finished dealing with the RPC.

gRPC: Server Program: MathServer

```
public class MathServerMain {

    public static void main(String[] args) {
        try {
            Server server = ServerBuilder
                    .forPort(50051)
                    .addService(new MathImpl()) // service implem
                    .build();

            server.start();
            System.out.println("Server started on port 50051");

            server.awaitTermination();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

gRPC: Server Program: MathServer

```
public class MathServerMain {

    public static void main(String[] args) {
        try {
            Server server = ServerBuilder
                    .forPort(50051)
                    .addService(    w MathImpl()) // service implem
                    .build();

            server.start();
            System.out.prin    ry           50051");

            s

        } ca


    }
}
```

We build and start our server using a ServerBuilder.

- Specify the address and port we want to use to listen for client requests using the builder's forPort() method.
- Create an instance of our service implementation class MathImpl and pass it to the builder's addService() method.
- Call build() and start() on the builder to create and start an RPC server for our service.

gRPC: Client Program: MathClient

```
public class MathClient {

    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder
                .forAddress("localhost", 50051)
                .usePlaintext().build();

        MathServiceGrpc.MathServiceBlockingStub stub =
                MathServiceGrpc.newBlockingStub(channel);

        BinaryOpRequest request = BinaryOpRequest.newBuilder()
                .setA(5).setB(3).build();

        BinaryOpResponse addResponse = stub.add(request);
        System.out.println("Add: " + addResponse.getResult());

        BinaryOpResponse multResponse = stub.mult(request);
        System.out.println("Mult: " + multResponse.getResult());

        channel.shutdown();
    }
}
```

gRPC: Client Program: MathClient

```
public class MathClient {

    public static void main(String[] args) {
        ManagedChannel channel = ManagedChannelBuilder
                .forAddress("localhost", 50051)
                .usePlaintext().build();

        MathServiceGrpc.MathSe    iceBlockingStub stub =
                MathServiceGrp    newBlockingStub(channel);

        BinaryOpRequest reques     BinaryOpRequest.newBuilder()
                .setA(5).          b

        Bina                                               .
        Syst                                               t());

                                                           ;
                                                           lt());

        C
    }
}
```

To call service methods, we first need to create a *stub*, a *blocking/synchronous* stub or a *non-blocking/asynchronous* stub.
First we need to create a gRPC *channel* for our stub, specifying the server address and port we want to connect to.
Then we can use the channel to create our stubs using the newStub and newBlockingStub methods provided in the MathServiceGrpc class we generated from our .proto.

gRPC: synchronous asynchronous

- **gRPC clients can choose** to call remote methods in **two ways**:
- Synchronous (Blocking) Calls:
  - The client sends a request and waits (blocks) until the server responds
  - Characteristics:
    - Simple and easy to use
    - Similar to a normal function call
  - Uses blocking stubs. Example: MathServiceBlockingStub
- Asynchronous (Non-blocking) Calls:
  - The client sends a request and does NOT wait for the response.
  - Characteristics:
    - Non-blocking → client can do other work
    - Uses callbacks, futures, or streams
    - Better performance for high-load systems
  - Uses async stubs. Example: MathServiceStub
  - Asynchronous programming is especially important for streaming modes because messages arrive over time

gRPC: types of RPC methods

- gRPC supports 4 kinds of service methods that can be defined in .proto file
- *Simple RPC*: client sends a request to server and waits for a single response to come back (unary). The normal function call
- *Server-side streaming RPC:* client sends a request to server and gets a stream to read a sequence of messages back. gRPC guarantees message ordering within an individual RPC call
- C*lient-side streaming RPC*: client writes a sequence of messages and sends them to server, waits for the server to read them and return its response. gRPC guarantees message ordering within an individual RPC call
- *Bidirectional streaming RPC*: both sides send a sequence of messages using a read-write stream (full duplex). gRPC preserves the order of messages in each stream

gRPC: Call semantics

- At-most-once (Default Behavior): a request is executed zero or one time
  - No duplicate executions (no accidental double processing)
  - the request might fail and never complete
  - Why:
    - gRPC runs over HTTP/2, which ensures ordered delivery but does not guarantee retry success
    - If a failure happens (timeout, network issue) client may not know if server executed the request or not
- At-least-once (Only with Retries Enabled): A request is executed one or more times
  - You enable retries in gRPC
  - Network failures trigger automatic resend
  - Same request may be executed multiple times => you must design your service to be Idempotent (safe to repeat)
  - Example: "add 100 to balance"
- Exactly-once: NOT guaranteed by gRPC

Source code

- MathGRPC.zip
- MathRMI.zip

Conclusions

- Remote Procedure Calls/Remote Method Invocations:
  - Simple way to pass control and data
  - Elegant transparent way to distribute application
  - Java RMI: very elegant and highly transparent solution, but obsolete
- Transparency vs performance: difficulties due to distributed nature of RPC/RMI (dealing with failures, performance)
  - Real solution: break transparency! Let programmer deal with these problems => gRPC
  - gRPC: cross-platform, scalable, performant communication for microservices and cloud
- Java RMI vs gRPC lessons learned:
  - Trying to be too transparent impacts negatively on performance
  - A solution confined to a single ecosystem will have limited usage