Distributed Objects

COMP 30220: Distributed Systems

Lecturer: Rem Collier

Email: rem.collier@ucd.ie

Introduction

- **Idea:** Distributed Objects are an Object-Oriented approach to implementing Remote Procedure Calls.
  - A system consists of a set of distributed objects.
  - Objects are identified by a unique identifier.
  - Computation involves the invocation of methods on specified objects.
  - This is done transparently whether the object is local or remote.
  - Distributed Objects Frameworks try to hide the network from the programmer.

- Distributed Object Frameworks:
  - CORBA – Common Object Request Broker Architecture
  - DCOM – Distributed Common Object Model
  - Java RMI – Java Remote Method Invocation
  - JINI – Apache Distributed Objects Project (Now Apache River)

Common Object Request Broker Architecture (CORBA)

Introduction to CORBA

- CORBA is a standard for enabling interaction between applications written in heterogenous languages.
  - Based on the definition of public interfaces written the Interface Definition Language (IDL).

```
module Finance {
  typedef sequence<string> StringSeq;
  struct AccountDetails {
    string name;
    StringSeq address;
    long account_number;
    double current_balance;
  };
  exception insufficientFunds { };
  interface Account {
    void deposit(in double amount);
    void withdraw(in double amount) raises(insufficientFunds);
    readonly attribute AccountDetails details;
  };
};
```

  - Language specific calls are mapped to the IDL interface:
    - Mappings exists for: C, C++, Ada, Cobol, Java, Lisp, Ruby, Python, …

IDL Mappings

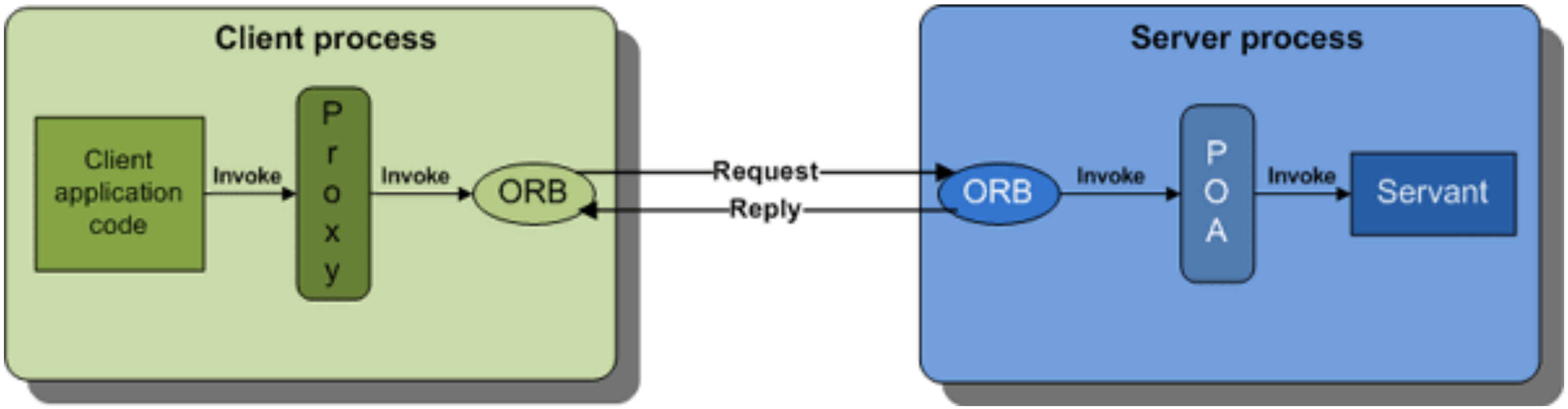
- Primitive types are mapped to their equivalent in the target language
  - This includes viewing `string` as a primitive type

- Other more complex types include:
  - `struct` types are mapped to C structs or classes containing only public fields.
  - `sequence` defines a vector type.
  - `typedef` is used to define new names for existing types
  - `union` defines a type that can hold several values at runtime
    - In Java, this is a class containing 1 field per type and a mechanism to check which type is returned – discriminator).
  - `interface` defines the distributed objects exposed by CORBA.
  - `module` defines a set of objects (like a package/namespace)

Using CORBA

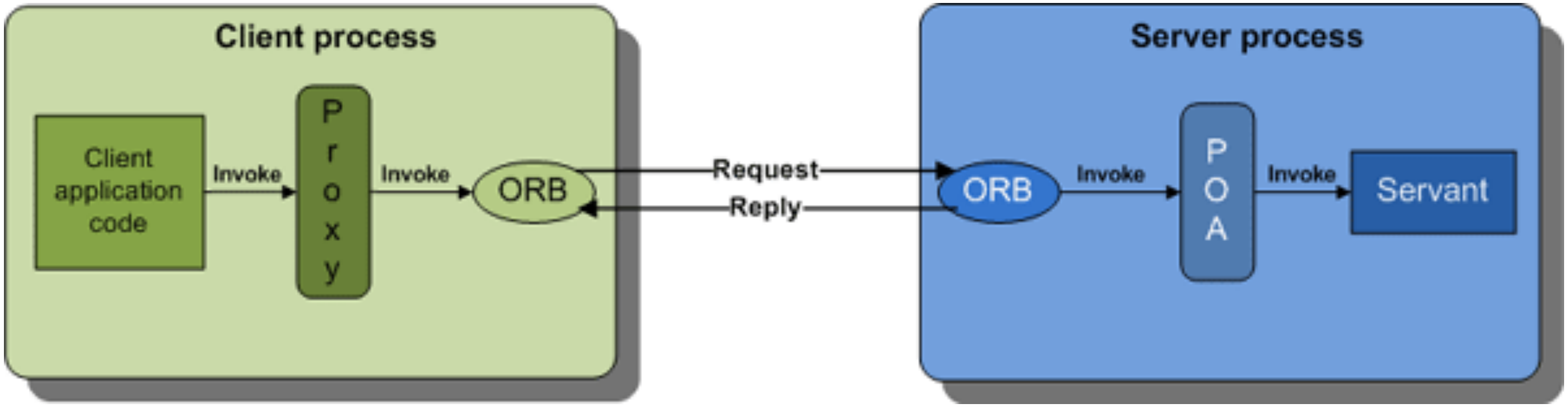
- Application development starts with an IDL definition.
- An IDL Compiler is used to generate code in a specific (set of) target language(s).
  - `structs` and `unions` are mapped to Java classes
  - Field types are inferred from primitive types, `typedefs`, `enums` and `sequences`.
  - The `interface` is mapped to a triplet of:
    - A **Java interface** of the same name
    - **POA(Portable Object Adaptor)/Skeleton code** that implements the server side of the network interface (and which is extended for specific implementations).
    - **Stub/Proxy code** that implements the client side (may not be needed).
  - All classes are organised within packages based on module names.

EXECUTING A CORBA RPC



- The Server creates deploys a CORBA object by:
  1. creating an ORB,
  2. creating a POA,
  3. creating the servant object,
  4. registering the POA/servant with the ORB, and
  5. advertising the POA/servant via the **COSNaming** service.

EXECUTING A CORBA RPC



- The Client uses a CORBA object by:
  1. locating the reference of the remote object via the COSNaming Service
  2. retrieving the proxy that represents the remote object
  3. Invoking a method on the proxy that causes the equivalent method on the remote object to be invoked.

IIOP: Internet Inter-ORB Protocol

- IIOP = GIOP + TCP/IP
- GIOP (General Inter-ORB Protocol) is an abstract protocol:
  - Defines byte-based data transfer syntax called Common Data Representation (CDR).
  - Supports 8 basic message types:
    - Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError, Fragment.
- IIOP does not support Fragment or MessageError
- IIOP is now used beyond the scope of CORBA

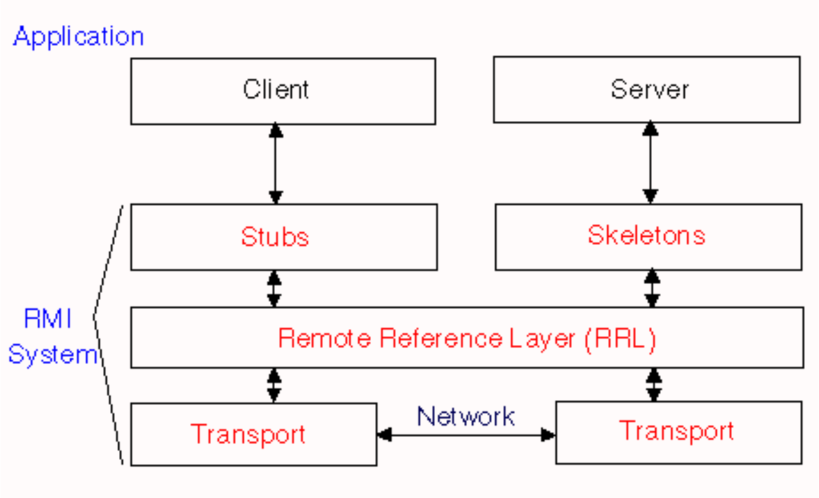
The Rise and Fall of CORBA

- CORBA pre-dates the rise of the Web.
  - In the 1990s, it was a leading technology for the creation of distributed applications.
- Its large suite of standards made the creation of fully standards compliant implementations difficult.
  - Full implementations were required in **each** target language.
  - Many implementations were considered poor.
  - Good implementations were licensed rather than open-sourced.
- Compared to newer technologies, CORBA was found to be lacking:
  - Code was not transferrable between ORBs
  - IIOP required exceptions to be added to firewalls (in contrast with SOAP)
  - Many projects did not require multi-language support and SOA/SOAP seemed easier for those that did.
  - CORBA-based systems often viewed as too brittle for Internet-scale applications.

Remote Method Invocation

Remote Method Invocation (RMI)

- Idea: A mechanism to allow Java objects in different JVMs to communicate "directly" with each other
  - A remote object lives on a server and has a remote interface that specifies which of its methods can be invoked by clients.
  - Local objects interact with the remote object by invoking methods on a local "stub" that deals with invoking the method on the remote object.
  - The stub and skeleton (the server side part of the remote connection) are created automatically.



Insert: Serialisation

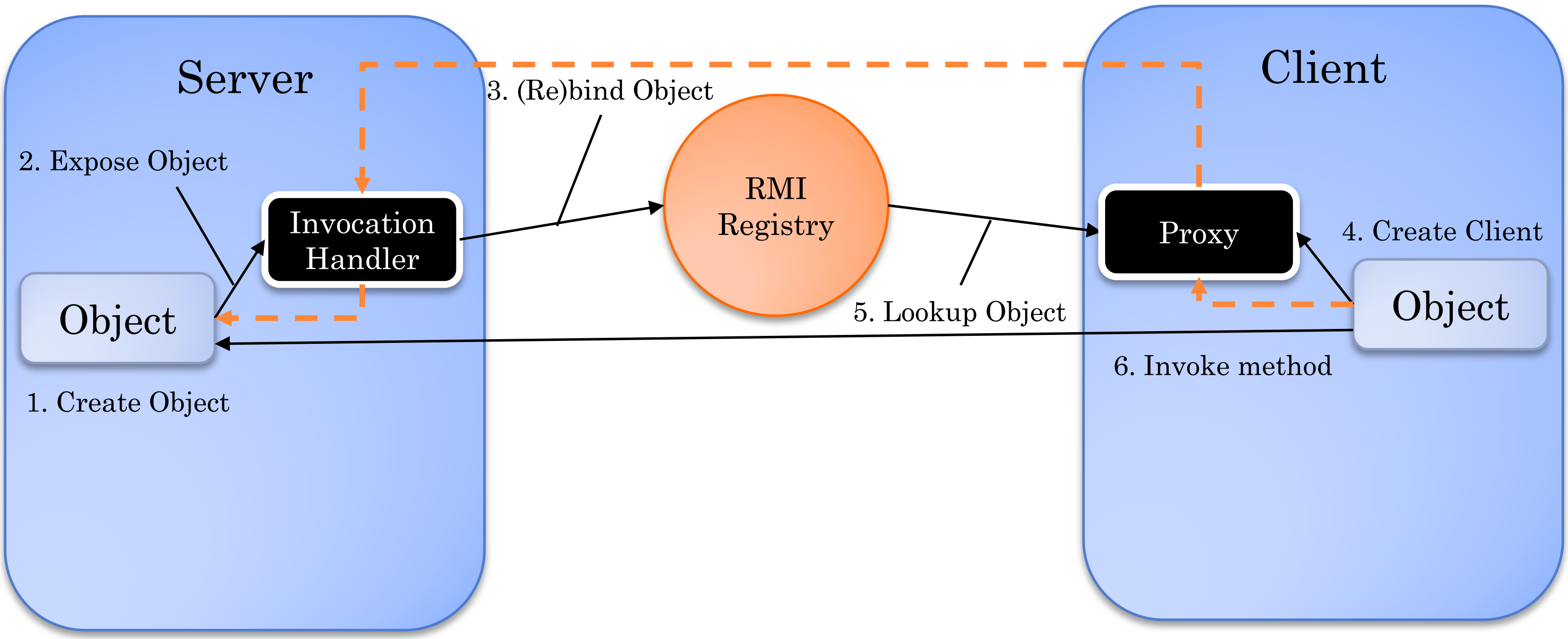
- Mechanism for converting an object into a stream of bytes:
  - Non-trivial as objects may be composed of other objects…
  - The stream of bytes can be sent anywhere: a file, a server, …
  - Once received, a copy of the original object can be recreated from the stream of bytes.
    - Actually, the destination device will also need the SAME class definition.

- Java supports serialization of primitive types and objects that implement the `java.io.Serializable` interface.
  - E.g. String, Vector, Integer, Float, Exceptions, most of AWT and Swing…
  - For full list, consult the Javadoc: http://docs.oracle.com/javase/9/docs/api/java/io/Serializable.html

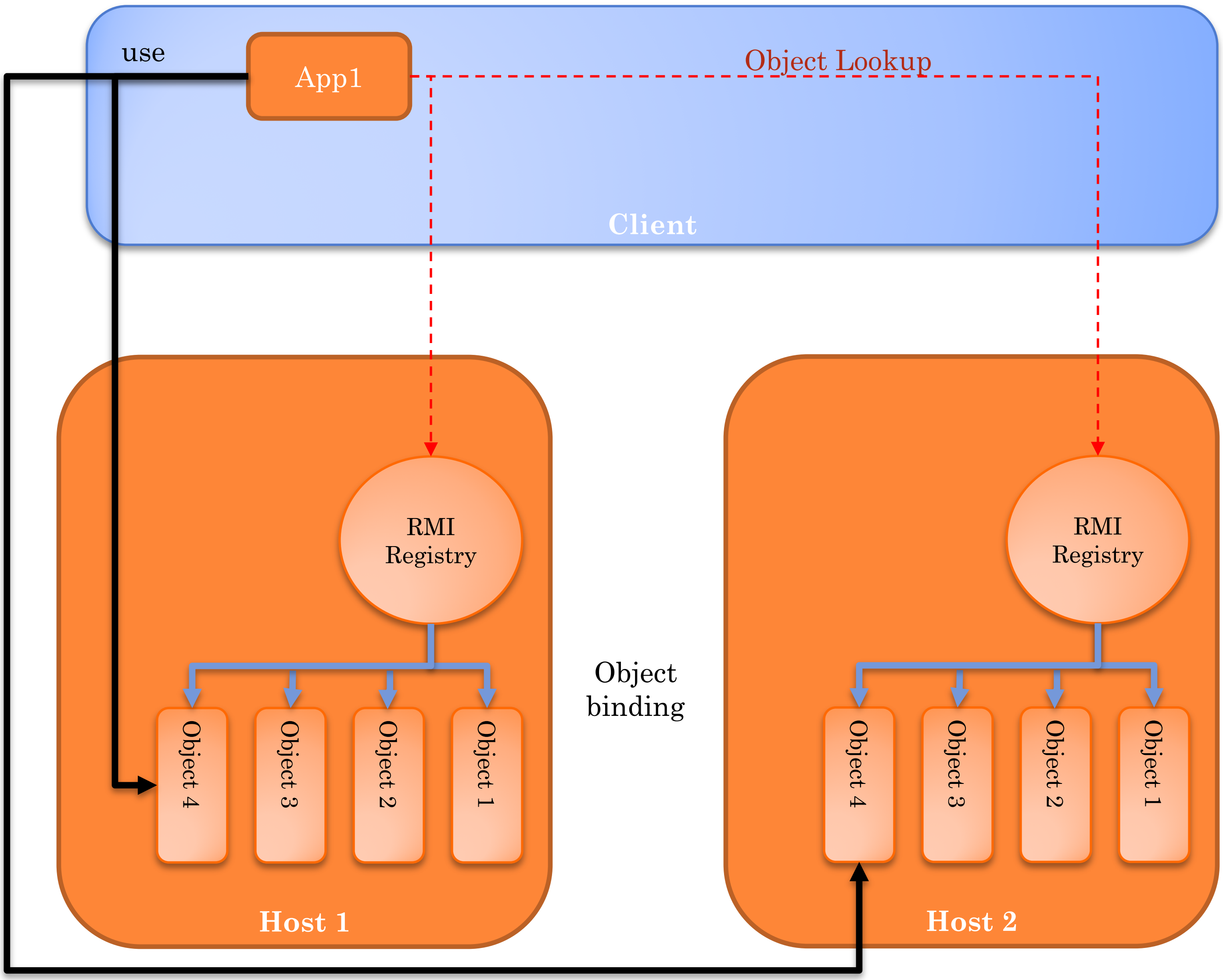
Java RMI Architecture

- **Key Design Principle**: Separate the implementation from the public interface.
  - Use Java interfaces to define the public interface of the remote object.
  - Implement the interface as a Java class.
- Export the implementation as a remote object.
  - Generates a **skeleton** to act as the remote object server.
    - Skeleton registers object with an RMI server (JRMP or IIOP)
  - Uses a **stub** (proxy) to act as the remote client.
    - Stub created using `java.lang.reflect.Proxy` class
    - Stub is a client that interacts with the skeleton.
- A global registry (naming service) is used to resolve object identifiers to remote object addresses (URLs).
  - Stub is stored in the registry and passed to the client for use

Java RMI Architecture



Java RMI Architecture



Example: RMI Calculator

RMI Calculator

- Basic Calculator Service:
  - Declare a Remote Object Interface:

```
public interface Calculator extends java.rmi.Remote {
  public long add(long a, long b) throws java.rmi.RemoteException;
  public long sub(long a, long b) throws java.rmi.RemoteException;
  public long mul(long a, long b) throws java.rmi.RemoteException;
  public long div(long a, long b) throws java.rmi.RemoteException;
}
```

  - Implement the Remote Object:

```
public class CalculatorImpl implements Calculator {
  public long add(long a, long b) { return a + b; }
  public long sub(long a, long b) { return a - b; }
  public long mul(long a, long b) { return a * b; }
  public long div(long a, long b) { return a / b; }
}
```

RMI CALCULATOR

- Basic Calculator Service:
  - Deploy the Remote Object:

```
public class CalculatorServer {
    public static void main(String args[]) {
        try {
            // Create the RMI Registry
            Registry registry = LocateRegistry.createRegistry(1099);

            // Create the Remote Object
            Calculator c = (Calculator)
                UnicastRemoteObject.exportObject(new CalculatorImpl(),0);

            // Register the object with the RMI Registry
            registry.bind("CalculatorService", c);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
}
```

RMI Calculator

- Basic Calculator Service:
  - Interact with Remote Object:

```
public class CalculatorClient {
    public static void main(String[] args) {
        try {
            // Get a reference to the RMI Registry
            Registry registry = LocateRegistry.getRegistry();

            // Find the distributed object (stub created here)
            Calculator c = (Calculator) registry.lookup("CalculatorService");

            // Do stuff!!!!
            System.out.println(c.sub(4, 3));
            System.out.println(c.add(4, 5));
            System.out.println(c.mul(3, 6));
            System.out.println(c.div(9, 3));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RMI Deconstructed

- The RMI Registry is a central server that stores information on the remote objects that are available.

- Creating the RMI registry:
  - You can run the RMI registry separately by executing the `rmiregistry` program in `bin` folder of the JDK.
  - Alternatively, you can create it programmatically via the following statement:

    ```
    Registry registry = LocateRegistry.createRegistry(1099);
    ```

- Getting the RMI registry:
  - Once the registry has been created, clients can access the registry via the following statement:

    ```
    Registry registry = LocateRegistry.getRegistry();
    ```

  - Alternatively:

    ```
    Registry registry = LocateRegistry.getRegistry("localhost",1099);
    ```

RMI Deconstructed

- Creating a Remote Service:
  - To do this, you simply create an instance of the Remote Service and pass it to the `exportObject(..)` method of the `UnicastRemoteObject` class.

```
Calculator c = (Calculator)
    UnicastRemoteObject.exportObject(new CalculatorImpl(), 0);
```

- Advertising a service:
  - This makes the service available over RMI.
  - To make a service available, you **bind** it to a name:

```
registry.bind("CalculatorService", c);
```

- Finding a service:
  - You need to search for a service with the same name as declared on the server:

```
Calculator c = registry.lookup("CalculatorService");
```

Full Registry API

- `void bind(String name, Remote obj)`
  - Binds the specified name to a remote object.
- `String[] list(String name)`
  - Returns an array of the names bound in the registry.
- `Remote lookup(String name)`
  - Returns a reference, a stub, for the remote object associated with the specified name.
- `void rebind(String name, Remote obj)`
  - Rebinds the specified name to a new remote object.
- `void unbind(String name)`
  - Destroys the binding for the specified name that is associated with a remote object.

RMI Limitations

- Only supports Java-to-Java interaction
- Lookup service limited to name.
  - What if you don't know the name?
  - What if you are looking for a type of service?
    - E.g. a weather service, a payment service, …
- Interaction is strongly coupled:
  - You decide on the object (service) you want to interact with.
  - Client blocks until the response is returned.
  - What happens if the service fails?
  - What if the service becomes overloaded?
- Remember: Distributed Systems are expected to handle failure and should be designed to scale with demand…

Note About the Proxy…

How Does the Proxy Work?

- RMI works because the stub and skeleton can be automatically generated.
  - This is achieved by using some magic from the **Reflection API**.
  - This API provides a set of classes for representing Java classes, Interfaces, Methods, Annotations, …
  - The most common use of the API is to create an instance of a class based on a string representation of its canonical name:

```
Object obj = Class.forName("java.lang.Object").newInstance();
```

  - One very interesting class in this API is the `java.lang.reflect.Proxy` class.
  - This class can be used to dynamically create an object that seems to be an instance of an explicitly specified set of Java interfaces – **even if no actual instance exists**.

How Does the Proxy Work?

```
public class DebugProxy implements java.lang.reflect.InvocationHandler {
    private Class cls;
    public static Object newInstance(Class cls) {
        return java.lang.reflect.Proxy.newProxyInstance(
                    cls.getClassLoader(), new Class[] {cls}, new DebugProxy(cls));
    }
    private DebugProxy(Class cls) { this.cls = cls; }
    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        System.out.println("class " + cls.getCanonicalName());
        System.out.println("method " + m.getName());
        System.out.println("args " + Arrays.asList(args));
        return 0l;
    }
}
public class Main2 {
    public static void main(String[] args) throws RemoteException {
        Calculator calculator = (Calculator) DebugProxy.newInstance(Calculator.class);
        calculator.add(50, 50);
    }
}
```

A Practical Example…

```java
public class DebugProxy implements java.lang.reflect.InvocationHandler {
    private Object obj;
    public static Object newInstance(Object obj) {
        return java.lang.reflect.Proxy.newProxyInstance(obj.getClass().getClassLoader(),
                        obj.getClass().getInterfaces(), new DebugProxy(obj));
    }
    private DebugProxy(Object obj) { this.obj = obj; }
    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        Object result;
        try {
            System.out.println("before method " + m.getName());
            result = m.invoke(obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        } catch (Exception e) {
            throw new RuntimeException("unexpected invocation exception: " + e.getMessage());
        } finally {
            System.out.println("after method " + m.getName());
        }
        return result;
    }
}
```

A Practical Example…

```
import java.rmi.RemoteException;

public class Main {
    public static void main(String[] args) throws RemoteException {
        Calculator calculator =
            (Calculator) DebugProxy.newInstance(new CalculatorImpl());
        System.out.println("Result: "+calculator.add(50, 50));
    }
}
```

```
----- EXPECTED OUTPUT -----

before method add
after method add
Result: 100
```

Apache River (AKA JINI)

Brief History

- Developed by Sun Microsystems using Java
  - First Released in 1998
  - Ownership transferred to Apache in 2012
  - Still active development - current release 3.0.0

- One of the first attempts to develop a decentralised infrastructure for Object-Oriented distributed systems.
  - Uses a custom protocol for remote method invocation (JERI).
  - Based around triplet of concepts: **Registry**, **Service**, **Client**.
  - Includes a HTTP server to support on-demand distribution of compiled code (**Class Server**).
  - Packaged with services for: locking, distributed events, leasing, transactions, etc.
  - Includes a **tuple space** implementation called **JavaSpaces**.
  - Highly configurable architecture

Lookup Service (LUS)

- Centralised service that consists of a set of **service items**:
  - **Unique identifier** is assigned to the service
  - **Service Type**: typically the fully qualified class name of the service.
  - **Associated set of attributes**: key-value pairs that define the properties of the service
  - **Client code**: downloadable interface to the service (e.g. RMI stub).

- Key Functionality:
  - Registration performed when a service is deployed.
  - Location of services can be based on id, type, attributes or some combination of the three.

Working a Service

- Each service consists of:
  - An implementation (in Java):
    - Implements `net.jini.lookup.ServiceIDListener`
    - Service creation and deployment logic implemented in constructor based on service configuration.
  - A service configuration file (implemented as named collections of key-value pairs)
    - Declares how the client is created
    - Sets necessary permissions for security
    - Declares static attributes (for the LUS)
    - Defines how the service can be discovered
- Clients use the LUS to locate a service, download the client code, and invoke the remote method.

JavaSpaces

- Tuple Spaces are a shared memory model for distributed systems.
  - Based on the Blackboard concept from Distributed AI
  - Data is shared between nodes in the form of **tuples –** a series of typed fields.
  - Nodes are able to read/write tuples from/to a shared space independent of their physical location.
  - When reading, tuples are selected via pattern matching – you specify some of the data (a tuple template) and the tuple space matches it to a tuple.

- In the JavaSpaces framework:
  - Objects are used as tuples (type matters here)
  - Templates are also objects, where **null** means **any value.**

PAPER ALLOCATIONS

- A – HADOOP
- B – SYNAPSE
- C – CASSANDRA
- D – CHUBBY
- E – ZOOKEEPER
- F – DYNAMO
- G – SCRIBE
- H – CEPH
- I – BIGTABLE
- J – MEGASTORE
- K – VOLDEMORT
- L – SPANNER
- M – PERCOLATOR
- N – ACOP
- O – AMBRY
- P – ESPRESSO
- Q – RAFT
- R – ACTORDBS
- S – SPARK
- T – KAFKA