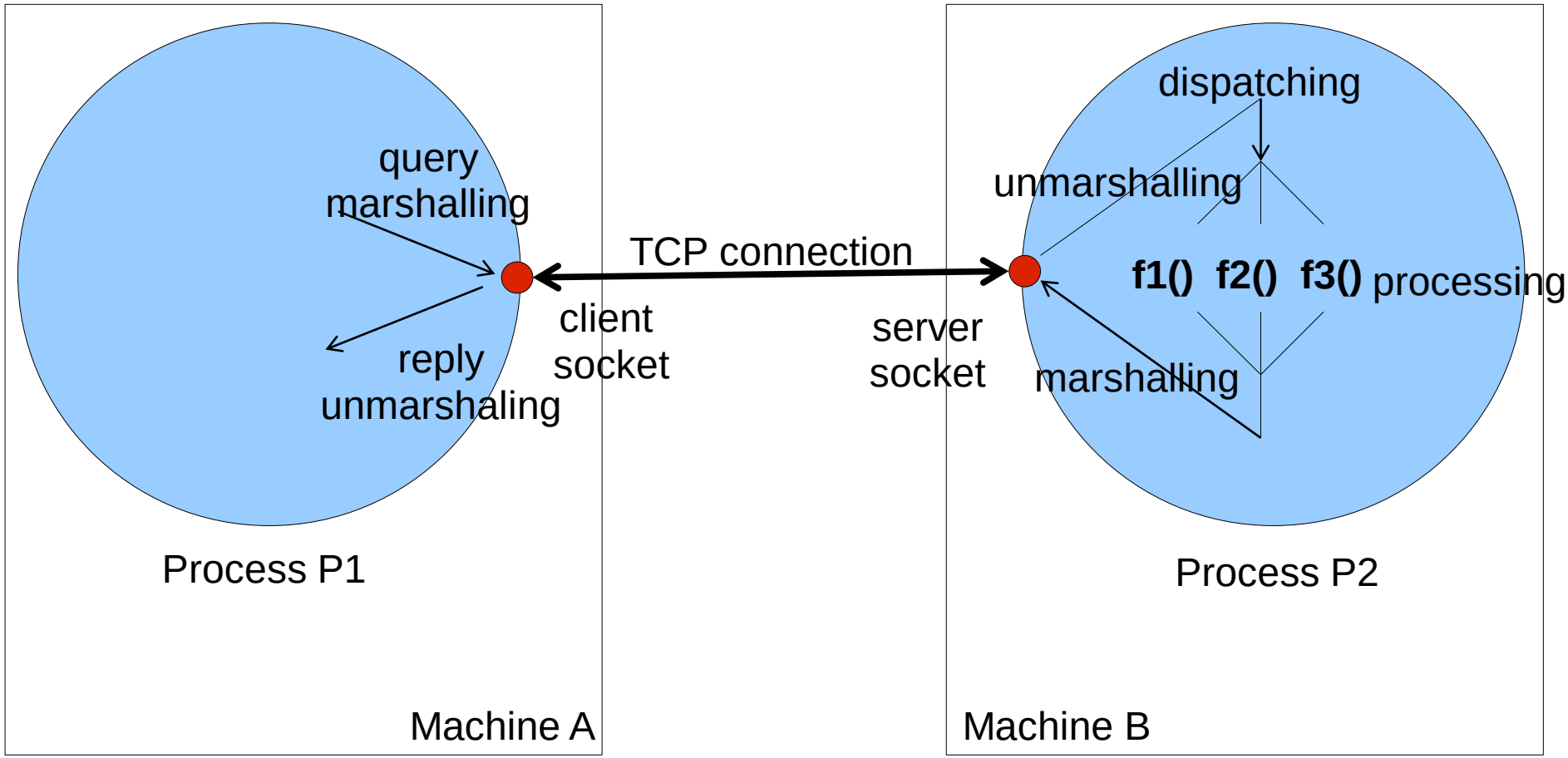
Remote Method Invocation (RMI)

MOTIVATION

Enterprise Applications

- Enterprise applications are often **distributed**, i.e. they consist of multiple processes
  - running on distinct machines
  - communicating with one another by message passing
- Java provides in the java.net package a networking infrastructure based on sockets, but
  - using this infrastructure is cumbersome (see next slide)
  - communications do not comply with the object paradigm
- By contrast, Remote Method Invocation provides
  - an easy-to-use, object-based infrastructure
  - object hosted by distinct JVM can call one another's methods as if they were hosted by the same JVM

Socket-Based Communication: Architecture



Socket-Based Communication: Drawbacks

- The developer has to handle all the low-level details of the communication. In particular, he has to:
  - establish and release the TCP connection
  - send, wait for and receive messages on both ends of the TCP connection
  - pack the values that make up a message on one end (this is called *marshalling*), and unpack them on the other end (*unmarshalling*)
  - dispatch the queries that arrives on P2 to the appropriate handling function f1(), f2(), etc.

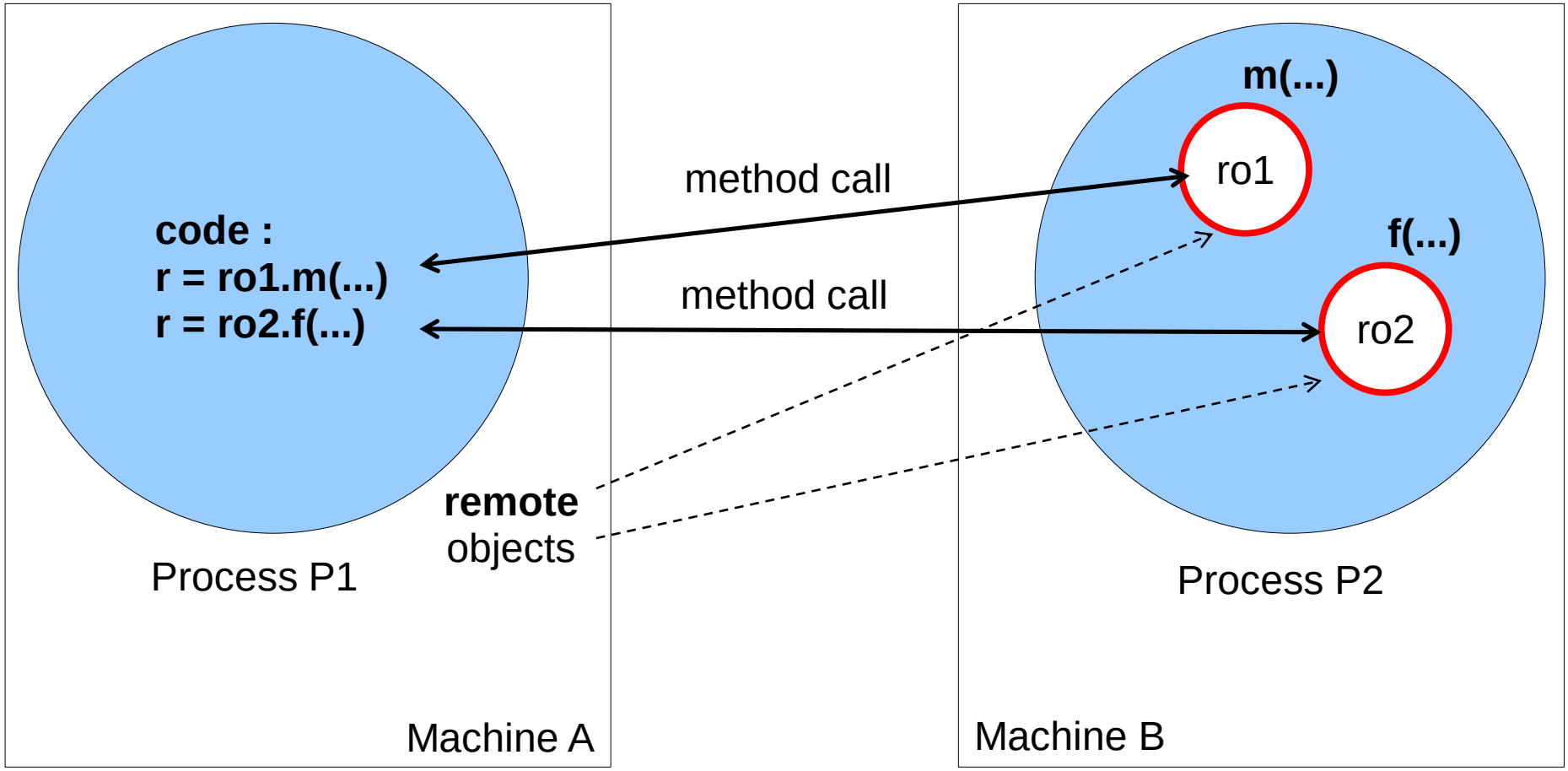
- This coding amounts to P1 calling some function of P2, but the object paradigm is not used

The Remote Method Invocation Solution

- Thanks to RMI, P1 sees P2 as a collection of objects ro1, ro2, etc.
  - these objects are said ***remote*** because they live in a JVM distinct from that of P1.
  - however, P1 can call the methods of these remote objects **as if** they were local, i.e. located in P1's JVM.

- RMI's runtime handle all the communication machinery needed to implement a remote method calls. RMI:
  - establishes and releases TCP connections between P1 and P2
  - marshalls the remote method's parameters into a query message
  - marshalls the remote method's return value into a reply message

Remote Method Invocation: Architecture



Remote Method Invocation: Advantages

- RMI relieves the developer from the burden of handling communication low-level details.
  - The developer can focus on the application code.
- RMI preserves the object paradigm in the world of distributed applications.
  - A distributed application is seen as a collection of (possibly remote) objects rather than a collection of processes.
- RMI achieves two key properties of distributed applications:
  - ***access tranparency***: remote objects are accessed in (almost) the same way as local objects
  - ***location transparency***: the location of the remote objects is (almost) hidden from the calling process

Notes

- RMI is the equivalent of Remote Procedure Call (RPC) in the world of Java objects.

- RMI vocabulary:
  - A process that creates a remote object is called a ***server***.
  - A process that calls a method on a remote object is called a ***client***.
  - A client may also create a remote object, in which case it will play the role of server for another client process.

IMPLEMENTATION

Skeleton

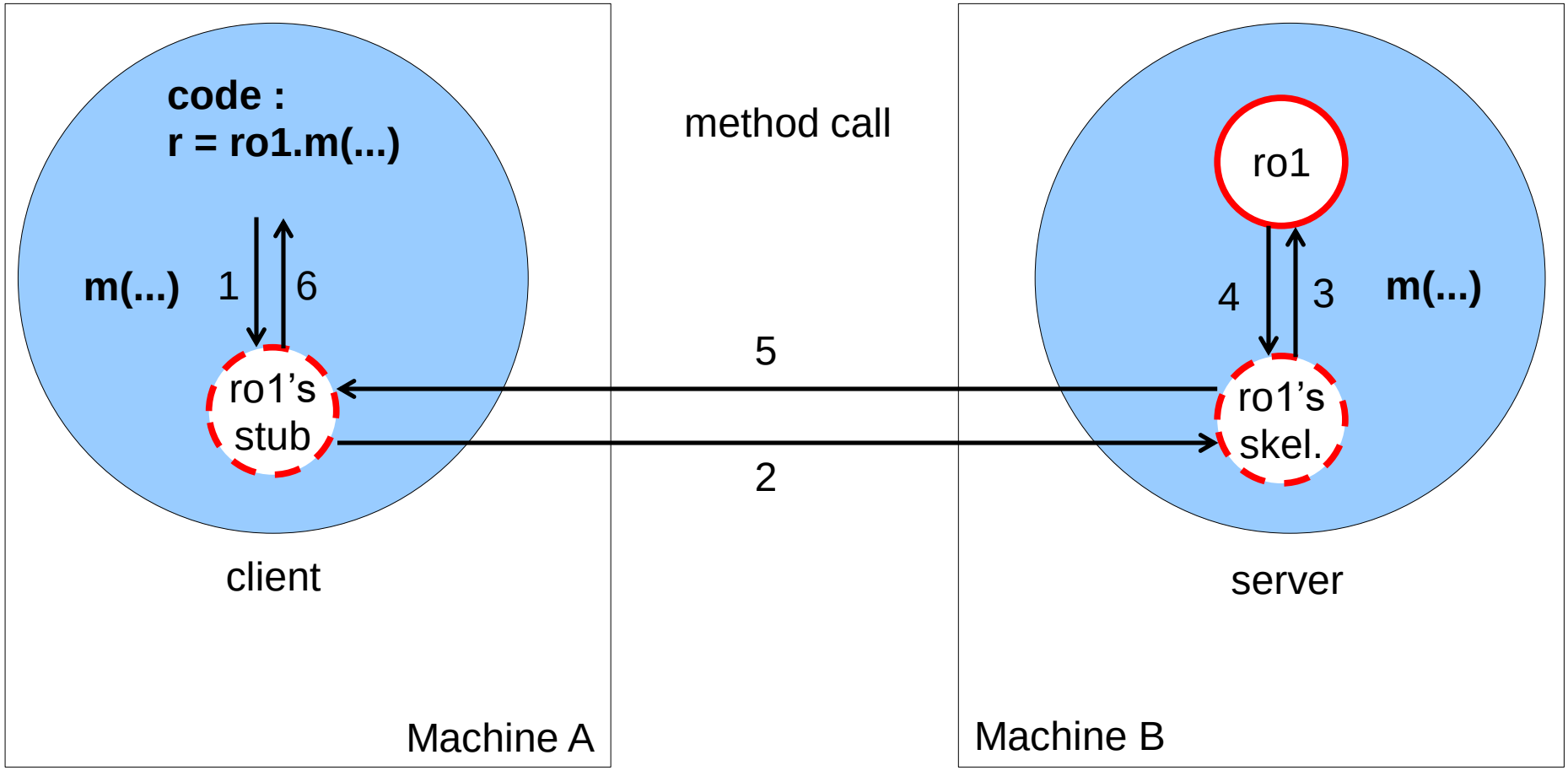
- When a server creates a remote object, RMI creates two additional objects: the ***skeleton*** and the ***stub***.

- The **skeleton** encapsulates the remote object on the server side and handles all communications to/from the remote object. The skeleton:
  - listens for incoming requests (i.e. method calls)
    - this is done by a listening thread
  - unmarshals method's parameters from the request
  - call the remote object's method, with the given parameters
  - marshals the method's return value into a reply message
  - sends the reply back to the client

Stub

- The stub is a local representative, or proxy, of the remote object on the client side. The stub has the same Java interface as the remote object and it contains information to locate the remote object's skeleton. The stub:
  - marshals the client call's parameters into a request
  - sends the request to the skeleton and waits for the reply
  - unmarshalls the reply and gets the call's return value
  - returns the value to the client code

Skeleton and Stub



Remote Object Designation

- Local objects are designated by a *reference* returned by the new operator. What about remote objects?
- Remote objects could be designated by the port number their skeleton listens on. But this is not flexible enough.
- Instead, remote objects are designated by a String ***name***, chosen by the application.
- Remote object names are well-known within an application, i.e. clients and servers know them.
  - just as port 21 is the well-known port to contact the ftp service

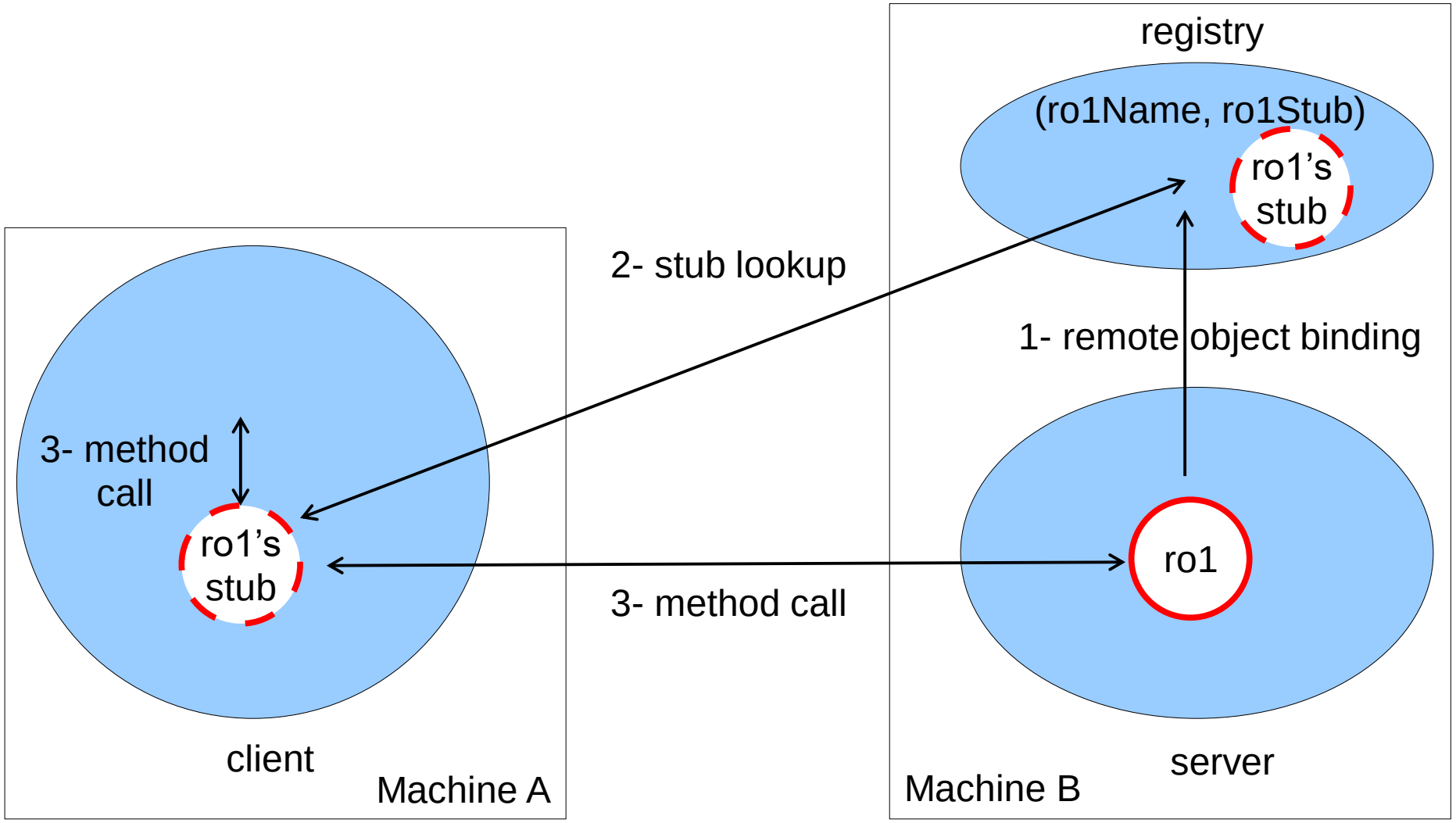
Registry

- The ***registry*** plays a central role in the RMI architecture. It stores (object name, object stub) associations. The registry is
  - updated by server processes, when they register remote objects: bind(), rebind() and unbind() methods
  - queried by client processes, when they need to access a remote object: lookup() method

- For security reasons, a registery cannot be updated by server processes running on remote machines.
  - one needs to run one registry on each server machine

- Note: the RMI registry is actually implemented as RMI remote object, which listens on a well-known port (default: 1099).

RMI Application Startup



DEVELOPMENT

RMI Application Development Steps

The development of an RMI application involves the following steps:

1. Develop the remote object's interface
2. Develop the remote object's implementation
3. Develop the server code
4. Develop the client code
5. Deploy the classes on the client and server machines

Example: We develop a **Sorter** service, which sorts List of String in ascending (**sort()** method) or descending order (**reverseSort()** method)

1- Sorter Remote Interface

```
public interface Sorter extends Remote {

  public List<String> sort(List<String> list) throws RemoteException;

  public List<String> reverseSort(List<String> list) throws RemoteException;

}
```

- For the RMI runtime to handle a remote interface, the interface must
  - extends the **Remote** interface
  - have all its methods throw **RemoteException**
- The RMI runtime will throw a RemoteException whenever a severe error, like a communication link failure, occurs.

2- Sorter Implementation

```
public class SimpleSorter implements Sorter {

  public List<String> sort(List<String> list) {
    Collections.sort(list);
    return list;
  }

  public List<String> reverseSort(List<String> list) {
    Collections.sort(list);
    Collections.reverse(list);
    return list;
  }

}
```

- Note: remote object's methods don't throw RemoteException; rather, this exception is thrown by the RMI runtime on the client side, when a communication error occurs.

3- Server Code

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

   // instanciate the remote object
   Sorter sorter = new SimpleSorter();

   // create a skeleton and a stub for that remote object
   Sorter stub = (Sorter) UnicastRemoteObject.exportObject(sorter, 0);

   // register the remote object's stub in the local registry
   Registry registry = LocateRegistry.getRegistry();
   registry.rebind("Sorter", stub);

}
```

- Note: the main method can also be declared in the SimpleTest class

4- Client Code

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

    // locate the registry that runs on the remote object's server
    Registry registry = LocateRegistry.getRegistry("zeus.google.com");
    System.out.println("client: retrieved registry");

    // retrieve the stub of the remote object by its name
    Sorter sorter = (Sorter) registry.lookup("Sorter");
    System.out.println("client: retrieved Sorter stub");

    // call the remote object to perform sorts and reverse sorts
    List<String> list = Arrays.asList("3", "5", "1", "2", "4");
    list = sorter.sort(list);

    list = Arrays.asList("mars", "saturne", "neptune", "jupiter");
    list = sorter.reverseSort(list);
}
```

- Note: the client must know (1) the name of the remote object, (2) the name of the machine that hosts it.

5- Deploying the classes

- We have the client machine and the server machine.
- On the client machine, we need to deploy:
  - the client class
  - the remote interface class
- On the server machine, we need to deploy:
  - the server classes
  - the remote interface class
  - the registry program
- Important: The Sorter interface needs to be deployed on **both** machines since it is referenced by the client and the server code. The interface must have the same fully qualified name (i.e. including packages) on both machines.

PASSING PARAMETERS

Aside: Serialization

- Serialization is a powerful Java feature. It allows an object to be converted into a byte array containing the values of the object's attributes.
  - Note: For security reasons, a class is not serializable by default; it must implement the **Serializable** interface for its instances to be serialized.
- Conversely, one can reconstruct (a copy of) the original object from the byte array. This is called deserialization.
- Note that the definition of the object's class (attributes name and type, methods, etc.) is not stored into the byte array.
  - Thus, the class definition of the object must be available in the JVM's class path during the serialization and deserialization process.

Uses of Serialization

- After an object has been serialized into a byte array, the byte array can either:
  - be stored in a file
  - or sent accross the network to a remote machine

- In the first case, serialization provides object **persistency**: an application can store its objects in a file before terminating; when restarted, the application can reconstruct its object in the last state they had.

- In the second case, serialization provides a mechanism for **copying** objects from one JVM to another.

Passing Parameters with RMI

- Local-object parameters are passed by ***value***. The stub sends a copy of each parameter to the skeleton using serialization / deserialization. Thus:
  - If the remote object modifies one of its local-object parameters, the corresponding parameter on the client side is not affected.
  - The class definition of the parameter should reside in the class path of the server's JVM.

- The same applies to the method 's return value: if the return value is a local object, it is passed by value from the server to the client.

Passing Parameters with RMI (2)

- Remote-object parameters are passed by ***reference***. Recall that a remote object is represented by its stub. The fact is a stub always designates the same remote object, even after serialization / deserialization. Thus:
  - If the remote object modifies one of its remote-object parameters, the modification will be seen on the client side.

- The same applies to the method's return value: if the return value is a remote object, it is passed by reference from the server to the client.

DYNAMIC CLASS LOADING

Motivation

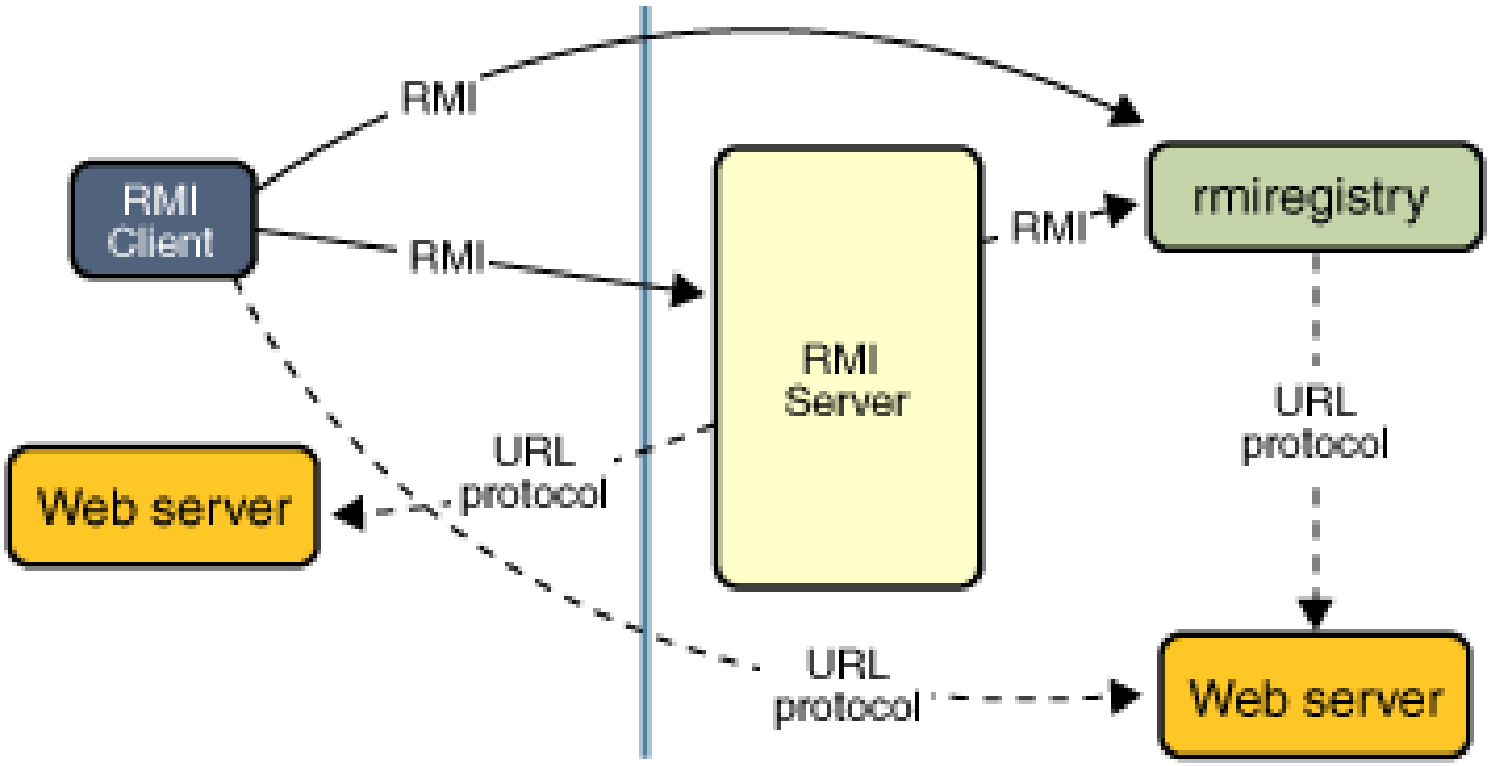
- The remote interface is well-defined and known by the client and the server before they are coded and deployed.
- However, the client can pass to the server a parameter whose class is not known to the server. Example:
  - the interface defines a paramater of type java.util.List
  - the client passes an object of class MyList, which implements java.util.List
  - the client has the definition of class MyList in its class path, but the server does not have it
- Consequence: the server will not be able to deserialize the MyList parameter when it receives it.
- The same problem can occur in the client if the server returns a value whose type is not known to the client.

Solution: Dynamic Class Loading over the Network

- To handle this case, the RMI runtime on the server side will
  1. download from the client the definition of class MyList over the network
  2. dynamically load the class definition in the server's JVM before deserializing the parameter

- To achieve this
  - the client provides an HTTP server that serves its *.class files
  - the stub of the remote object includes the following information in the messages it sends to the skeleton:
    - the class name of each parameter
    - the URL of the HTTP server to contact if the class is missing in the server

Overall Architecture



Security

- The Java language was designed for safety. By default, the JVM is not allowed to download and execute classes from remote computers.
- Thus, for dynamic class loading to work, the developer has to instruct the JVM's security manager to:
  - allow remote classes dowloading
  - allow remote classes execution
- This configuration is specified in a security property file