

# Remote Method Invocation



Benoît Garbinato  
distributed object programming lab

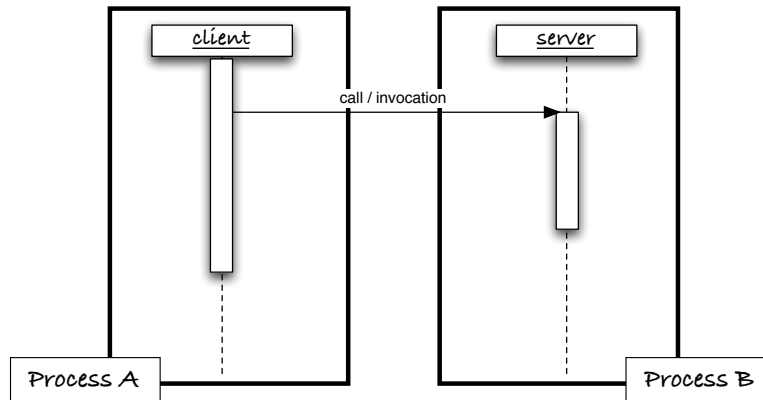
1

## Fundamental idea (1)

- Rely on the same programming paradigm for distributed applications as for centralized applications
  - In procedural languages, we will rely on the notion of Remote Procedure Call (RPC)
  - In object-oriented language, we will rely on the notion of Remote Method Invocation (RMI)

2

## Fundamental idea (2)

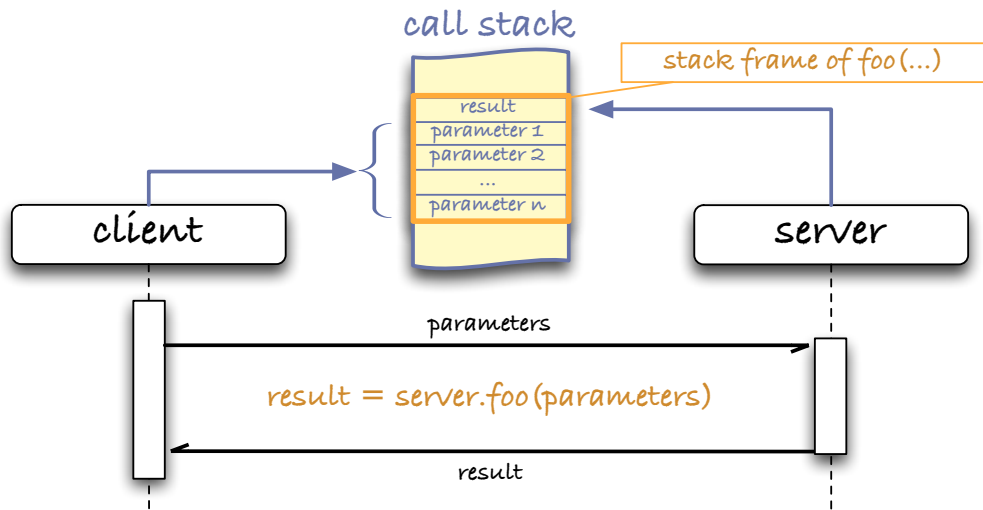


A remote method (procedure) is transparently invoked (called) across the network, as if it was local

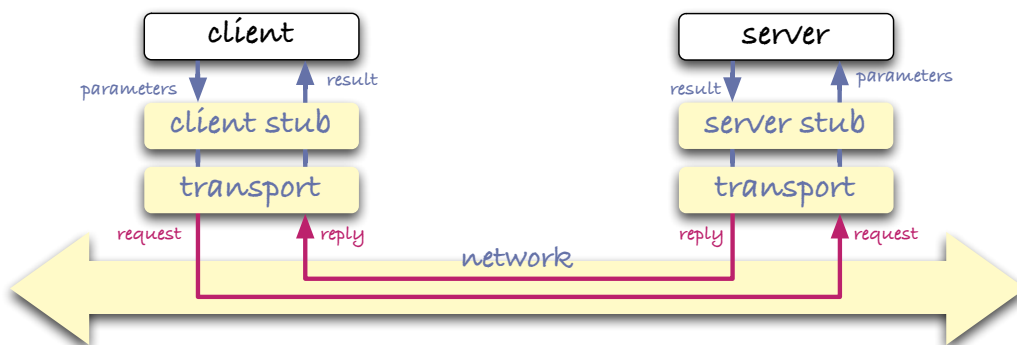
## RPC: some history

- 1979 Bill Joy introduces the "Berkeley Enhancements", mainly interprocess communication (IPC) facilities. The modern network Unix is born (BSD).
- mid 80's Sun Microsystems uses BSD Unix as operating system for their workstations. They extend it with RPC, on top of which they build NFS and NIS (later on NIS+).
- 1988 The Open Software Foundation (OSF) is formed to develop a portable open system platform, known as the Distributed Computing Environment (DCE). The latter proposes DCE RPC as basic communication mechanism.
- mid 90's The Object Management Group (OMG) follows the same approach to devise the Common Object Request Broker Architecture (CORBA) for object-based middleware. At the same time, Sun greatly simplifies & extends the RMI paradigm its Java & Jini platforms.
- Today Everybody talks about Web Services as the next "big thing", but it is merely a web-flavored version of the RPC/RMI paradigm, using HTTP & XML.

# A local method invocation

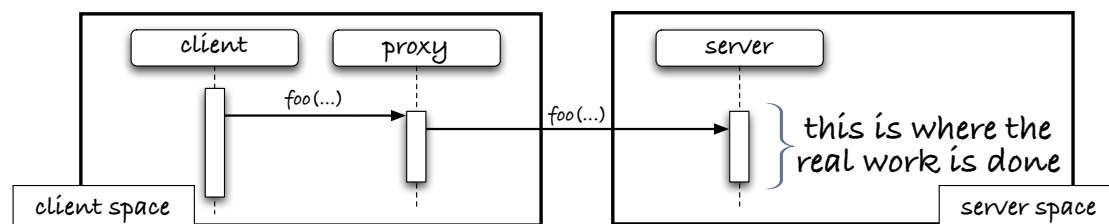


# A remote method invocation



# The notion of proxy

- A proxy is the representative of a server object in the address space of the client
- A proxy implements the same interface as the server (but not in the same way)



Remote Method Invocation © Benoît Garbinato

dop i a b

7

# Java RMI

- In Java, Remote method invocation is integrated in the standard class library, via packages such as `java.rmi`, `java.rmi.server`, etc.
- In addition, Sun's Java Development Kit (JDK) includes a set of tools for supporting RMI, e.g., `rmic`, `rmiregistry`, etc.
- We can distinguish three distinct times when building rmi-based applications, namely development, deployment and execution.

Remote Method Invocation © Benoît Garbinato

dop i a b

8

# Execution time

1. The server object registers its name & proxy in the naming service (rmi registry)
2. The client object obtains a proxy of the server object via that naming service
3. The client object can then invoke the server proxy, which will then forward the invocation to the server object

# Server side: create & bind

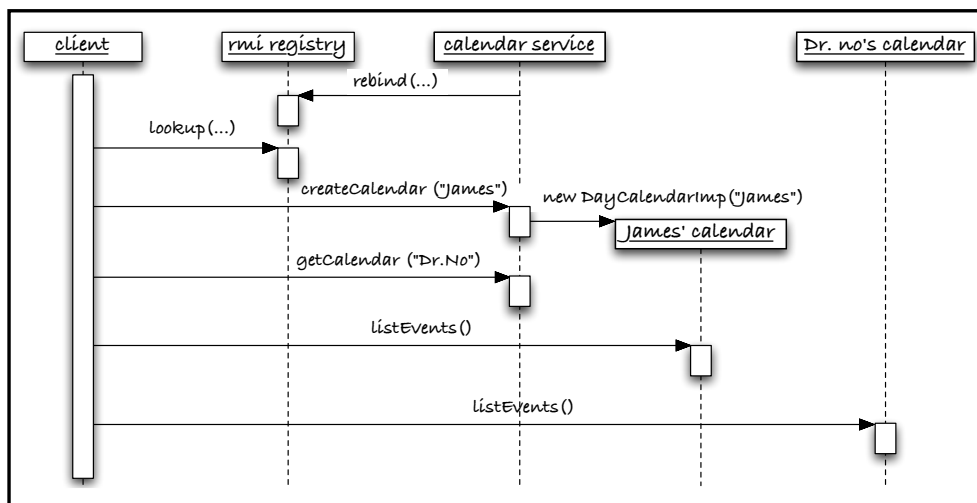
```
public class CalendarApp {  
    ...  
    public static void main(String[] args) throws Exception {  
        String theName= "Calendar";  
        CalendarServer theServer=new CalendarServer();  
  
        Naming.rebind(theName, theServer);  
        System.out.println("Calendar service is running!");  
    }  
}
```

# Client side: lookup & use

```
public class CalendarClient {
    ...
    public static void main(String[] args) throws Exception {
        String calServName= "://www.acme.com/Calendar";
        CalendarService calServ=
            (CalendarService) Naming.lookup(calServName);

        calServ.createCalendar("James");
        Collection allCals= calServ.getCalendars();
        DayCalendar dno= calServ.getCalendar("Dr. No");
        String[] elist= dno.listEvents();
    }
}
```

# Calendar Application



# Development time

1. Define the interface of the remote service
2. Implement the client and server classes in a decoupled way, thanks to the interface
3. Use javac to compile all above sources
4. Use the rmic compiler to create the proxy of the remote class for you

# Typical remote interfaces

```
import java.util.*;
import java.rmi.*;

public interface CalendarService extends Remote {
    public DayCalendar createCalendar(String name) throws RemoteException, CalendarException;
    public DayCalendar getCalendar(String name) throws RemoteException, CalendarException;
    public ArrayList getCalendars() throws RemoteException;
    public boolean exists(String name) throws RemoteException;
}
```

```
import java.util.*;
import java.rmi.*;

public interface DayCalendar extends Remote {
    public boolean isFree(Date date) throws RemoteException;
    public DayEvent plan(DayEvent event) throws RemoteException, CalendarException;
    public String[] listEvents() throws RemoteException;
    public String getName() throws RemoteException;
}
```

# A typical remote class

```
public class CalendarServer extends UnicastRemoteObject implements CalendarService {
    private Hashtable calendars;

    public CalendarServer() throws RemoteException {
        calendars= new Hashtable();
    }
    public DayCalendar createCalendar(String name) throws RemoteException, CalendarException {
        if (calendars.containsKey(name)) throw new CalendarException(name + "\" already exists.");
        DayCalendar newCal= new DayCalendarImpl(name);
        calendars.put(name, newCal);
        return newCal;
    }
    public DayCalendar getCalendar(String name) throws RemoteException, CalendarException {
        if (!calendars.containsKey(name)) throw new CalendarException(name + "\" does not exist.");
        return ((DayCalendar) calendars.get(name));
    }
    public ArrayList getCalendars() throws RemoteException {
        return new ArrayList(calendars.values());
    }
    public boolean exists(String name) throws RemoteException {
        return calendars.containsKey(name);
    }
}
```

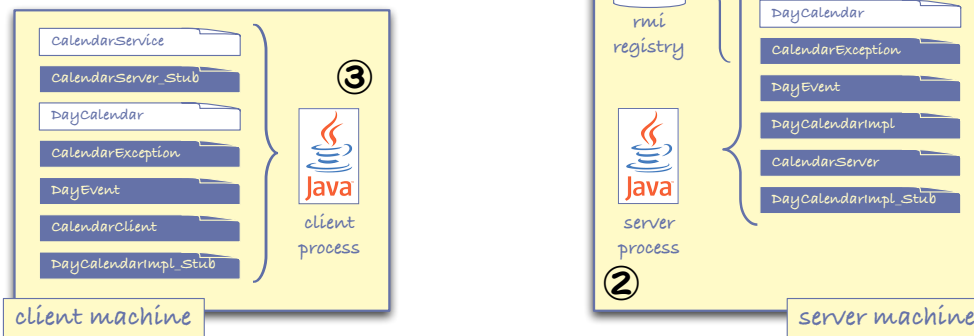
# Argument passing rules

1. An argument or a return value can be a primitive type, a local serializable object (i.e, implementing `java.io.Serializable`), or a remote object.
2. A primitive type value is passed by copy.
3. A local object is also passed by copy, using standard object serialization.
4. A remote object is passed by reference, i.e., its proxy is passed rather than the object itself.



# Deployment time

1. Start the rmi registry
2. Start the server process
3. Start the client process



# Checkup

- On the server we have:  
`CalendarServer theServer= new CalendarServer();`  
whereas on the client we have:  
`CalendarService calServ=`  
`(CalendarService) Naming.lookup(calServName);`  
why this difference?
- Where are calendars located?
- How does the client get access to calendars?
- How do we communicate with the rmi registry ?

# RMI callbacks (1)

- A remote object does not need to be registered in the naming service to be remotely accessible, e.g., DayCalendarImpl.
- The client can also make an object remotely accessible to the server, allowing the latter to asynchronously call back the client, e.g., to notify the client that a new event was scheduled on some calendar.

# RMI callbacks (2)

```
public interface CalendarListener extends Remote {
    public void eventPlanned(DayEvent e) throws RemoteException;
}

public interface DayCalendar extends Remote {
    public boolean isFree(Date date) throws RemoteException;
    public DayEvent plan(DayEvent event) throws RemoteException, CalendarException;
    public String[] listEvents() throws RemoteException;
    public String getName() throws RemoteException;
    public void addListener(CalendarListener l) throws RemoteException;
}

public class CalendarClient extends UnicastRemoteObject implements CalendarListener {
    ...
    public static void main(String[] args) throws Exception {
        String calServName= "//www.acme.com/Calendar";
        CalendarService calServ= (CalendarService) Naming.lookup(calServName);
        DayCalendar dno= calServ.getCalendar("Dr. No");
        CalendarListener calist= new CalendarClient();
        dno.addListener(calist);
        dno.plan(new DayEvent(new Date(), "Conquer the world"));
    }
    public void eventPlanned(DayEvent e) throws RemoteException {
        System.out.println("--> New event planned: " + e);
    }
}
```

# RMI callbacks (3)

```

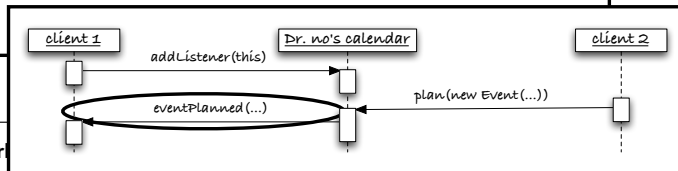
public class DayCalendarImpl extends UnicastRemoteObject implements DayCalendar {
    private TreeSet eventSet;
    private ArrayList listeners;
    ...

    public void addListener(CalendarListener l) throws RemoteException {
        listeners.add(l);
    }

    private void notifyListeners(DayEvent e) {
        Iterator iter= listeners.iterator();
        while ( iter.hasNext() )
            try {
                ((CalendarListener) iter.next()).eventPlanned(e);
            } catch (RemoteException re) { System.err.println("Notification failed"); }
    }

    public DayEvent plan(DayEvent event) throws RemoteException, CalendarException {
        if (eventSet.contains(event)) throw new CalendarException("The date is not free");
        eventSet.add(event);
        notifyListeners(event);
        return event;
    }
}

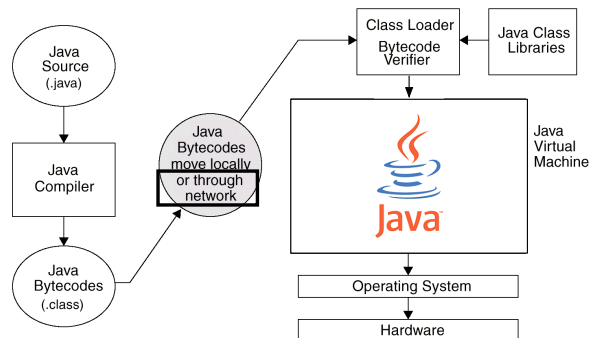
```



Remote Method Invocation © Benoît Garbinato

# Dynamic code download (1)

- The Java platform allows for the dynamic download of classes from any URL (Uniform Resource Locator)

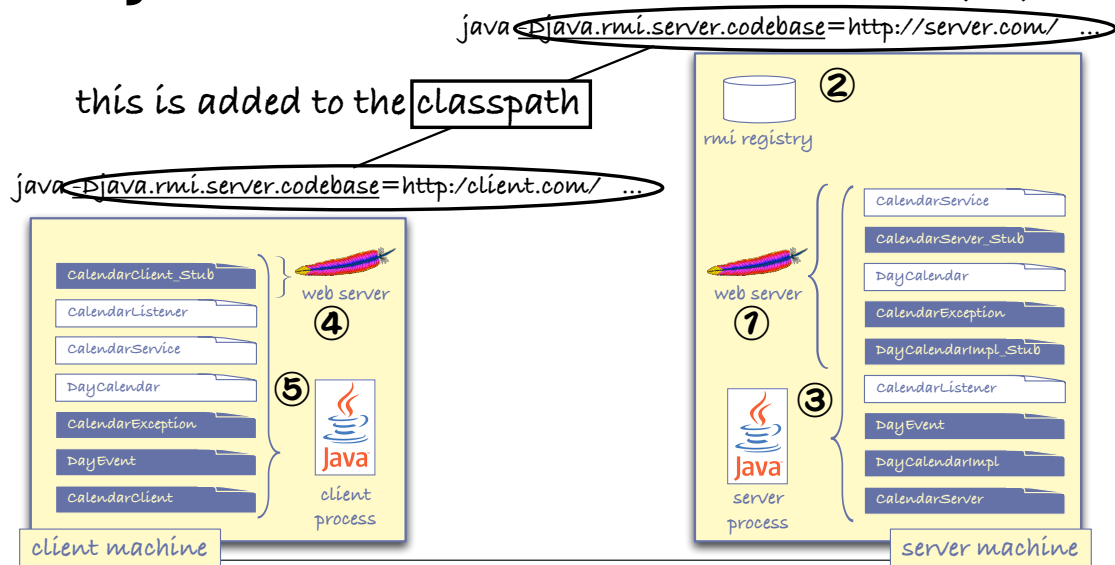


Remote Method Invocation © Benoît Garbinato

## Dynamic code download (2)

- The proxy is located on the client but it conceptually belongs to the server
- Because we have a Java virtual Machine on both the server and the client, it is possible to have the proxy move from the server to the client at runtime (dynamic code download)
- Dynamic code download can be used not only for proxies but for any Java class

## Dynamic code download (3)



# Security viewpoint

- From a security viewpoint, downloading classes is a critical action (i.e., potentially dangerous)
- For this reason, when code download is activated (via the `java.rmi.server.codebase` property), the Java virtual machine requires a security manager to be installed
- The security policy enforced by the security manager can be expressed declaratively in a security policy file

# Security manager & policy

Source code:

```
if (System.getSecurityManager() == null)
    System.setSecurityManager(new RMISecurityManager());
```

Command line:

```
java -Djava.security.policy=my.policy ...
```

Policy files:  
(`my.policy`)

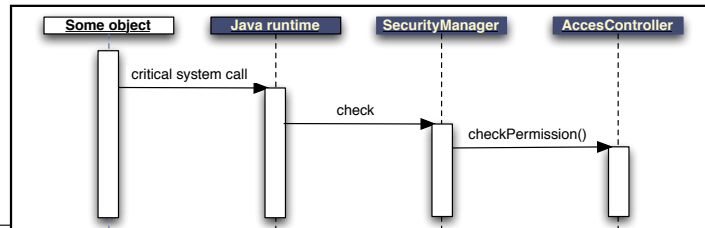
```
grant {
    permission java.net.SocketPermission "server.com:1024-65535", "connect,accept";
    permission java.net.SocketPermission "server.com:80", "connect,accept";
};
```

```
grant {
    permission java.net.SocketPermission "*:1024-65535", "connect,accept";
    permission java.net.SocketPermission "*:80", "connect,accept";
};
```

client

server

Runtime:



# Distributed Garbage Collection

- The Java platform transparently extends garbage collection to distributed objects. This extension is known as Distributed Garbage Collection (DGC).
- A remote object is collected when there no longer exists any remote or local references to it
- Any object referenced by the naming service (rmi registry) is not collected

# Unreferenced vs. finalized

- By implementing the Unreferenced interface, a remote object can ask to be notified when there no longer exists any remote references to it
- In the unreferenced() method, the remote object is given the opportunity to release some resources, e.g., the remote reference on a another remote object

```
public class DayCalendarImpl extends UnicastRemoteObject implements DayCalendar, Unreferenced {
    ...
    public void unreferenced() {
        System.out.println("-> Oups, I am no longer remotely referenced!");
    }
    protected void finalize() throws Throwable {
        System.out.print("This time, I am really about to be garbage collected...");
        System.out.print("so bye bye cruel world!");
    }
}
```

— called by the distributed garbage collector

— called by the local garbage collector

# Limitations of DGC

- An implementation of DGC should ensure
  - Safety, which implies not collecting too early
  - Liveness, which implies eventually collecting
- Due to its inherent decentralized nature, the implementation of DGC is based on reference counting, which poses several issues:
  - It does not deal properly with circular references
  - It does not deal properly with asynchronous systems
- Partial solution: the notion of lease

# The notion of *lease*

- A lease is a remote reference with a validity limited in time
- In Java, remote references are actually leases
- If the client does not renew its lease before the associated timeout expires, the reference counter on the server side is decremented
- Leases are automatically managed for you, i.e., the renewal is automatic as long as the client is alive and the remote reference exists

**Questions?**