Remote Invocations

Hans-Arna Jacobsen
jacobson@eecg.toronto.edu
www.eecg.toronto.edu/~jacobsen

amount = 1000;
account.withdraw(amount);

- what is a remote invocation?
- how can the integer amount be transferred over the network, be re-assembled on the server side?
- does the client wait for a response from the server?
- what if the network goes down while the invocation is in progress?
- how is this programmed? ... in Java? in XYZ?
- what tools exist to do remote invocations?
  - our examples: CORBA, Java RMI, and Web Services (current trend)
- what are the limitations of remote invocation, are there any at all?
Outline (2 weeks)

- Remote invocation concepts
- Case studies
  - Java RMI
  - CORBA
  - Web Services
- A look under the hood (presentation formatting et al.)
- summary and comparison

Today's and next times lecture draws material from:

A concise summary of the CORBA architecture.
Sun Java RMI. Company white paper.
Read between the lines to learn about RMI, lots of hype included.
Web Services papers (short overview statements). IBM Developer Works.
A very short overview, very little information content, for lack of a better paper.

Thanks to Charles Zong and Linfeng for preparing an earlier version of these slides.

Local invocations & procedure calls

(local) method invocation (OO-paradigm):

\[
\text{return}_{-}\text{amount} = \text{account}.\text{withdraw}(20);
\]

- the message 'withdraw(20)' is sent to object account

\[
\text{withdraw}(20) \quad \text{account}
\]

(local) procedure call (imperative paradigm):

\[
\text{return}_{-}\text{amount} = \text{withdraw}(\text{account}, 20);
\]

- remote invocation protocol designs have aimed at carrying the analogy of a (local) invocation to the distributed context, i.e., making a call to a distributed node, should just be like making a local call.
- however, what is the difference between a centralized and a distributed system?
Centralized systems (non-distributed)

- one computer system with non-autonomous parts (however, may be a multi-CPU system)
- computer system shared by users at all time
- all resources accessible, managed by operating system (i.e. single point of control)
- software runs in a single process, however inter-process communication is possible
- single point of failure (i.e., computer system either runs or does not run, no intermediate state possible (e.g., no partial failure, due to network break down etc.)
- one contiguous addressable memory
- calling overhead is very low, we distinguish
  - address space local calls, i.e., method invocation
  - inter-address space, i.e., process-to-process communications

Distributed systems

- multiple autonomous computer systems (nodes)
- computer systems are not shared by all users, i.e., different users interact with different nodes at any one point in time
- not all resources may always be accessible to any interacting entity
- software runs in concurrent processes on different processors on different nodes, i.e., multiple points of control
- any part of the system may fail independently (nodes, network), i.e., multiple points of failure and partial failure are possible
- each node manages its memory, therefore no contiguously addressable memory (exceptions are shared memory systems, shared data spaces, not covered in this lecture)
- calling overhead may be considerable, i.e., call has to go out in the network, has to reach remote system, competes with other requests, and has return
Differences

- in the distributed case failure has to be managed
  - timing out of invocations
  - re-tries of invocations
- data structures that are transported across the network
  have to be re-created at the remote node, but how should
  pointers be treated (e.g., linked lists)?
- invocations may take longer, may this effect our system, e.g.,
  do parts of our system require timely responses, otherwise,
  they, themselves may take actions (e.g., time-out of network
  connections, releasing of locks etc.)
- concurrency becomes an issues, shared data has to be
  protected
- some of these issues also apply to single nodes that run
  multiple processes concurrently or in parallel
- middleware platforms address these issues to varying
  degrees

Remote invocation

- What is a remote invocation?
  Remote invocation is a mechanism (i.e. a protocol) to allow the calling
  of services (of procedures or of methods) over a network of physically
  distributed computer systems.

  We often say a client invokes a method on a server, i.e. client/server
  computing paradigm.

  A server may realize a number of different services (may implement a number
  of different objects).

- Why do we need a remote invocation protocol?
  1. Programming networking code is fairly complex and error prone.
  2. In a distributed computing environment, the locations of the server objects
     are often not known or changing. Location transparency is desired.

- Often referred to as
  - message passing (in the 70/80ss)
  - RPC - remote procedure call (imperative programming paradigm)
  - RMI - remote method invocation (object oriented programming paradigm)
  - Services - web services (service & component oriented programming)

  Different terminology, but overall same concepts.
Addressing a remote node

- an IP address identifies a node (a computer) on the network
- any number of networked applications may run on each node, each is identified by a port number, e.g., port 80 for http traffic etc.
- an (IP address, port number)-pair thus identifies a networked application uniquely
- a socket is an operating system service that allows applications to send and receive messages over a network
- a socket is a rather low-level abstraction
- many programming languages provide libraries to use sockets for network programming
- with vanilla sockets it is difficult to achieve the transparency of making a distributed call like a local call - on the other hand this may not be desirable at all (see difference bw. centralized and distributed sytems)

Remote invocation simplifies networking programming complexity

Before

Descriptor = socket(AEINET, SOCK_STREAM, 0)

Connect(descriptor, Receiving Address)

Write(descriptor, "data", Datapacket)

"data"

After

Client Procedure

parameters

results

Server Procedure

Descriptor=socket(AEINET, sock_stream,0)

Bind(descriptor, receiving Address)

Listen(descriptor, maxQueue(onm))

newDescriptor=accept(descriptor, Sending Address);

BytesReceived=reform(descriptor, Buffer, dataSize);
Some points to discuss

- pass-by-reference
- pass-by-value
- passing objects (i.e., state and behavior)
- serializing objects
- deep-copy

Synchronous Vs Asynchronous Invocation

- **Synchronous** invocation
  In the context of remote invocation, the client execution stops after invoking a remote method. The execution resumes after receiving results from the remote objects.

- **Asynchronous** Invocation
  Client execution continues its execution without waiting for the results. The client execution gets the results either via polling or interruption later in the execution.
How does remote invocation work?

A remote invocation mechanism often involves the following steps:

Server Side:
- Declaration of server operations.
- Implementation of server operations.
- Publication of the reference of the remote service.

Client Side:
- Obtain client side stub/proxy code for marshaling/de-marshal- ing (unmarshaling)
- Obtain the remote object reference and invoke like "local" programs
How is Remote Invocation done?

Marshaling and unmarshaling

expServer.getPolicy();

<table>
<thead>
<tr>
<th>Marshal parameters</th>
<th>Send Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unmarshal parameters</td>
<td>Invoke implementation</td>
</tr>
</tbody>
</table>

impl

return new Today'sPolicy();

skel

Receive return (or exception)

Marshal return (or exception)

Send Reply

stub

Unmarshal reply

Return value (throw exception)

FIGURE 3 Stubs and Skeletons
Remote Invocation Nomenclature

- **Interface Definition**: The server side needs to provide definitions of its operations for the client side to invoke. The definitions are often written using an interface definition language.

- **Marshaling**: A process that transforms high level data structures into sequences of bytes in order to transmit them over the network.

- **Unmarshaling**: The reverse process of marshaling. Reconstruction of high level data structures from the received byte sequences that come in over the networks.

- Often (un-)marshaling also subsumes packing and unpacking, not just the encoding step (e.g., higher-level data structures first have to be packed before they can be encoded for delivery over a network)

*What are these higher-level data structures?*

Remote Invocation Nomenclature

- **Stub**: Client side code that serves as the proxy of the remote services. It contains marshaling and unmarshaling (packing and unpacking) code (often referred to as proxy).

- **Skeleton**: (server stubs) provides interfaces to each operation or service exported by the server. Same as client stubs, it contains marshaling and unmarshaling code, packing and unpacking, and upcalling (dispatching) code as well (often referred to as stub or proxy).

- Stubs and skeletons are often automatically generated from the interface definition of the server.

- The **stub-compiler** generates these stubs and skeletons from the interface definitions.
Remote Invocation By Example

- Remote invocation mechanism is the foundation of many distributed technologies.
- We are going to look at the implementation of remote invocation using the following technologies:
  - Java RMI
  - CORBA
  - Web services

Some points that came up in the reading (distributed) garbage collection:

- A programming language system that manages its memory by itself is called garbage collected - memory that is no longer used is de-allocated; Java is garbage collected (so is Lisp), C and C++ are not garbage collected (e.g., user has to free memory explicitly)

Design pattern:

- Refers to a commonly accepted way of designing a software system, often at the implementation level (i.e., at the coding level); examples are the observer pattern, the factory pattern, the factory pattern

Factory:

- An implementation abstractions for creating an instance (an object) of a given class, or creating a number of instances that belong together
Some points cont.'d

wrapper:
- a piece of code that translates between a "published" application programming interface (API) and an "unpublished" different API, e.g.,
  - an object-relational wrapper of a relational database
  - a Java wrapper of a legacy API of a COBOL system
  - a wrapper that allows a program to submit queries to a search engine; extract stock quotes for a web site etc.

- this is an example of a design pattern - wrapper or translator pattern

Java RMI

The following slides up to "Activation in Java", have been adapted from slides by Ian Gorton, Paul Greenfield Software Architectures and Component Technologies, CSIRO, Sydney.
RMI overview

- in Java 1.0 object communication confined to objects in one Virtual Machine
- Remote Method Invocation (RMI) introduced in Java 1.1 supports communication between different VMs, potentially across a network
- RMI is tightly integrated with Java language
- minimise changes to Java language and to VM
- work in homogeneous environment, but offers support for native language integration (JNI – Java native language integration, i.e., for C)

Java Object Model

- interfaces
- classes
- Java objects as instances of classes
- attributes (state of objects)
- operations (behaviour of objects)
- exceptions
- inheritance (code and interface)
- remote objects
Java Interfaces and Remote Objects

- Java already includes the concept of interfaces
- RMI does not have a separate interface definition language (e.g., DCE RPC or CORBA have)
- Pre-defined interface Remote
  - Remote interfaces extends Remote
  - Remote classes implement remote interfaces
  - Remote objects are instances of remote classes
- stubs and skeletons are generated from server implementation

Java Remote Interface Example

```java
package soccer;
interface Team extends Remote {
    public:
    String name() throws RemoteException;
    Trainer[] coached_by() throws RemoteException;
    Club belongs_to() throws RemoteException;
    Players[] players() throws RemoteException;
    void bookGoalies(Date d) throws RemoteException;
    void print() throws RemoteException;
}
```

Remote operations
Attributes

- RMI does not have attributes
- attributes must be represented as set and get operations by the designer
- Example:

```java
interface Club extends Organization, Remote {
    public:
    int noOfMembers() throws RemoteException;
    Address location() throws RemoteException;
    Team[] teams() throws RemoteException;
    Trainer[] trainers() throws RemoteException;
    ...
};
```

Attribute get operations

Combining Classes and Remote Interfaces

```java
interface Organization {
    private:
    String name() RemoteException;
};
class Address {
    public:
    String street;
    String postcode;
    String city;
};
interface Club extends Organization, Remote {
    public:
    int noOfMembers() throws RemoteException;
    Address location() throws RemoteException;
    Team[] teams() throws RemoteException;
    Trainer[] trainers() throws RemoteException;
    void transfer(Player p) throws RemoteException;
};
```

Club can return an address object
Club makes name() remotely accessible
Parameter Passing

- atomic types are **passed-by-value**
- remote objects are **passed-by-reference**
- Non-Remote objects are passed by value

class Address {
    public:
        String street;
        String postcode;
        String city;
};
interface Club extends Organization, Remote {
    public:
        Address location() throws RemoteException;
        ...
};
returns a copy of the address!

Exception

- Pre-Defined Exception `RemoteException`
- Type-Specific Exceptions
- Example:

  Type-specific Exception

  class PlayerBooked extends Exception {};
  interface Team extends Remote {
      public:
          ...
          Operation declares that it may raise it
          void bookGoalies(Date d) throws 
              RemoteException, PlayerBooked;
      ...
  };
Architecture

Client

Stub

Registry Interfaces

Skeleton

Activation Interfaces

RMI Runtime (rmid, rmiregistry)

Activation in Java

Stub

Faulting Reference

Live ref Activation ID

1: activate

4: update live ref

Client Host

Java VM 1

AG 1

2: create object in VM

3: pass object ref

Java VM 2

AG 2

Activator

Activation Descriptors:
ActGroup ClassName URL Init
AG 1 Team www.bvb.de/...
AG 2 Player www.bvb.de/...
AG 2 Player www.bvb.de/...
AG 2 Player www.bvb.de/...

Host www.bvb.de
RMI by example: Calculator

• Our remote application is a calculator, which provides an add operation that returns the result of the sum of two numbers.
• We model the application using the class diagram below

```
public interface Calculator extends java.rmi.Remote {
    public int add (int number1, int number2) throws java.rmi.RemoteException {
        return number1 + number2;
    }
}
```

```
public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject implements Calculator {
    public int add (int number1, int number2) throws java.rmi.RemoteException {
        return number1 + number2;
    }
}
```

Java RMI development steps

Step 1: Interface Definition
   public interface Calculator extends java.rmi.Remote{
       public int add (int number1, int number2) throws java.rmi.RemoteException
   }

Step 2: Provide server side implementation
   public class CalculatorImpl extends java.rmi.server.UnicastRemoteObject implements Calculator {
       public int add (int number1, int number2) throws java.rmi.RemoteException {
           return number1 + number2;
       }
   }
Step 3: Generate stubs and skeletons

```
olive:~/src> rmic CalculatorImpl
```

Two files generated:

```
Calculator_Stub.class  Calculator_Skel.class
```

Step 4: Publish identity (through naming service)

```
Calculator c = new CalculatorImpl();
Naming.rebind("rmi://localhost:1099/Calculator", c);
```

Step 5: Client Look up and Invoke

```
Calculator c = (Calculator)
    Naming.lookup("rmi://localhost/Calculator");
c.add(1,2);
```

Case study I:
Passing behavior with Java RMI
Online expense reporting system

- employees enter their (travel) expenses online
- client GUI displays fields to be completed by user
- client communicates via RMI with server
- server stores entered information in a DBMS
- this is a typical multi-tiered application
- design for change!
- Example:
  - today: receipts for items above $20 are required
  - in the future: receipts are required for all purchases, except meals that cost less than $20

Expense reporting architecture

*FIGURE 1  An Expense Reporting Architecture*
How could these requirements be implemented?

- install policy with client
  - policy changes, all clients need to be updated
- policy could be checked by server upon update of each entry in the expense report
  - generate lots of network traffic, one round trip per field
  - not resilient to network failure
- policy could be checked by server upon submission of entire report
  - errors can only be signaled to the user at the end
- downloadable policies, i.e., client checks whether policy has changed, if so, downloads it and uses it
  - requires a trusted environment and the ability to download code

Dynamic policies: I.e. downloading of policy

- client connects to server and asks whether policy has changed
- server, either replies with no, or with the new policy object
- client downloads the new policy object and uses it to check the entries performed by the user locally
Advantages of the dynamic policy scheme

- clients don't need to be halted and updated with new software
- software is updated on the fly on a by-need basis
- server is not burdened with entry checking that can be done locally.
- allows dynamic constraints because object implementations, not just data, are passed between client and server.
- lets users know immediately about errors.

Interfaces

- remote interface available to client (i.e., we say services or methods offered by server):
  import java.rmi.*;
  public interface ExpenseServer extends Remote {
    Policy getPolicy() throws RemoteException;
    void submitReport(ExpenseReport report)
      throws RemoteException, InvalidReportException;
  }

- "local" policy interface, implemented by objects on the client machine, i.e., in client's VM
  public interface Policy {
    void checkValid(ExpenseEntry entry)
    throws PolicyViolationException;
  }
**Client code using remote and “local” objects**

Policy curPolicy = server.getPolicy();

*start a new expense report*

*show the GUI to the user*

**while** (user keeps adding entries) {

**try** {

curPolicy.checkValid(entry);
// throws exception if not OK

*add the entry to the expense report*

} **catch** (PolicyViolationException e) {

*show the error to the user*

}

server.submitReport(report);

---

**Server code**

```java
import java.rmi.*;
import java.rmi.server.*;
class ExpenseServerImpl
{
    extends UnicastRemoteObject
    implements ExpenseServer
    {
        ExpenseServerImpl()
            throws RemoteException {
            // ...set up server state...
        public Policy getPolicy(){
            return new TodaysPolicy();
        public void submitReport(ExpenseReport report) {
            //...write the report into the db...
    }
```

(c) 2002 Hans-Arne Jacobsen (jacobson@eceg.toronto.edu) MIE991
A policy example

```java
public class TodaysPolicy implements Policy {
    public void checkValid(ExpenseEntry entry)
        throws PolicyViolationException {
        if (entry.dollars() < 20) {
            return; // no receipt required }
        else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException; }
    }
}
```

Another policy example, may replace the former

```java
public class TomorrowsPolicy implements Policy {
    public void checkValid(ExpenseEntry entry)
        throws PolicyViolationException {
        if (entry.isMeal() && entry.dollars() < 20) {
            return; // no receipt required }
        else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException; }
    }
```
Case study II:
Passing behavior from client to server

Compute Server

FIGURE 2  An Compute Server Architecture
Local and remote interfaces

- models a generic compute task

```java
public interface Task { Object run(); }
```

- models the remote compute server

```java
import java.rmi.*;
public interface ComputeServer
    extends Remote {
        Object compute(Task task)
            throws RemoteException;
    }
```

- client creates a compute task and sends it for execution to server

```java
import java.rmi.*;
publi c interface ComputeServer extends Remote {
    Object compute(Task task)
        throws RemoteException;
}
```

Compute server implementation

```java
import java.rmi.*; import java.rmi.server.*;
public class ComputeServerImpl
    extends UnicastRemoteObject
    implements ComputeServer {
    public ComputeServerImpl() throws RemoteException { }
    public Object compute(Task task) {
        return task.run();
    }
}
```

```java
public static void main(String[] args) throws Exception{
    // use the default, restrictive security manager
    System.setSecurityManager(new RMISecurityManager());
    ComputeServerImpl server = new ComputeServerImpl();
    Naming.rebind("ComputeServer", server);
    System.out.println("Ready to receive tasks");
    return;
}
```
Case study III: Connecting to an existing server

Defining an interface for the remote server

```java
import java.rmi.*;
import java.sql.SQLException;
import java.util.Vector;
public interface OrderServer extends Remote {
    Vector getUnpaid()
        throws RemoteException, SQLException;
    void shutDown()
        throws RemoteException;
    // ... other methods (getOrderNumber,
    // getShipped, ...)
}
```
Accessing the remote (DBMS) server via JDBC

```java
import java.rmi.*; import java.rmi.server.*;
import java.sql.*; import java.util.Vector;

public class OrderServerImpl
   extends UnicastRemoteObject
   implements OrderServer
{
   Connection db; // connection to the db
   PreparedStatement unpaidQuery; // unpaid order query

   OrderServerImpl() throws RemoteException, SQLException {
      db = DriverManager.getConnection("jdbc:odbc:orders");
      unpaidQuery = db.prepareStatement("...");
   }

   public Vector getUnpaid() throws SQLException {
      ResultSet results = unpaidQuery.executeQuery();
      Vector list = new Vector();
      while (results.next())
         list.addElement(new Order(results));
      return list;
   }

   public native void shutDown();
}
```

Native method access via JNI

```c
JNIEXPORT void JNICALL Java_OrderServerImpl_shutDown(JNIEnv *env, jobject this){
   jclass cls;
   jfieldID fid;
   DataSet *ds;
   cls = (*env)->GetObjectClass(env, this);
   fid = (*env)->GetFieldID(env, cls, "dataSet", "J");
   ds = (DataSet *) (*env)->GetObjectField(env, this, fid);
   /* With a DataSet pointer we can use the original API */
   DSshutDown(ds);
}
```
Common Object Request Broker Architecture (CORBA)

Part of the following slides on CORBA, have been adapted from slides by Ian Gorton, Paul Greenfield Software Architectures and Component Technologies, CSIRO, Sydney.

Who is the OMG?

- Non-profit organization with HQ in the US, representatives worldwide.
- Founded April 1989.
- More than 800 members.
- Dedicated to creating and popularizing object-oriented industry standards for application integration, e.g.
  - CORBA
  - ODMG
  - UML
Goal of CORBA

- Support distributed and heterogeneous object request in a way transparent to users and application programmers
- Facilitate the integration of new components with legacy components
- Open standard that can be used free of charge
- Based on wide industry consensus

Object Management Architecture

Application Objects

Domain Interfaces

CORBA facilities

Object Request Broker

CORBA services
Object Model and Interface Definition

- Objects
- Types
- Modules
- Attributes
- Operations
- Requests
- Exceptions
- Subtypes

OMG Interface Definition Language

- Language for expressing all concepts of the CORBA object model
- OMG/IDL is
  - programming-language independent
  - oriented towards C++
  - not computationally complete
- Different programming language bindings are available
- Many changes from version 2.2 to 2.3 and 3.0
Basic Example

Clients hold references to objects

CORBA Object Model: Objects

- Each object has one identifier that is unique within an ORB
  - like a phone number
- Multiple clients can hold references to same and different objects
- References support location transparency
  - machine/process transparency
- Object references are persistent
  - stored and retrieved from disk
typedef struct _Address {
    string street;
    string postcode;
    string city;
} Address;

typedef sequence<Address> AddressList;

interface Customer { ... };

module MyOrg {
    typedef struct _Address {
        string street;
        string postcode;
        string city;
    } Address;
};

module People {
    typedef struct _Address {
        string flat_number;
        string street;
        string postcode;
        string city;
        string country;
    } Address;
};
CORBA Object Model: Attributes

```java
interface Customer;
typedef sequence<Customer> CustList;
interface Supplier;
typedef sequence<Supplier> SuppList;
interface E-Org {
    readonly attribute string name;
    attribute CustList customers;
    attribute SuppList suppliers;
};
```

CORBA Object Model: Operations

```java
interface Org {
    ... void getStockPrice(in Date d);
    string printDetails();
};
```
CORBA Object Model: Requests

- Requests are defined by client objects
- Request consist of
  - Reference of server object
  - Name of requested operation
  - Actual request parameters
  - Request is executed synchronously

```
Org myOrgObj;
//Initialize myOrgObj to refer to a remote //Org object - more later
myOrgObj->printDetails();
```

Parameter Passing

- Rules govern how in and out parameters are passed (more later..)
- Until CORBA 2.3, CORBA objects can only be passed by reference, not value
  - i.e a reference to a remote object as a parameter
  - need to manually deconstruct and construct an object by defining a struct to hold the object’s data
- Value Types now support passing objects by value
- This course won’t cover value types...

(c) 2002 Hans-Arne Jacobsen (jacobson@eceg.utoronto.ca) MIE491F
CORBA Object Model: Exceptions

- Generic Exceptions (e.g. network down, invalid object reference, out of memory)
- Type-specific Exceptions

```
Exception name     Exception data
exception OutOfStock{sequence<Date> since;}
interface Org {
  ...
  short order(in Date d, in Stock s)
    raises(OutOfStock);
};
```

Operations declare exceptions they raise

CORBA Object Model: Subtypes

```
Implicit supertype:
Object

interface Organization {
  readonly attribute string name;
};
interface OpUnit : Organization {
  exception NotEnoughCash{};
  readonly attribute short noOfStaff;
  readonly attribute Address location;
  void transfer(in Cash p) raises NotEnoughCash;
};
```

Implicit supertype: Object

Inherited by OpUnit

Supertype
One standardised interface
One interface per object operation
One interface per object adapter
ORB-dependent interface

**Key Points**

- CORBA, COM and RMI
  - enable objects to request operation execution from server objects on remote hosts
  - identify server objects by object references
  - distinguish between interface and implementation
  - treat attributes as operations
  - provide mechanisms to deal with failures
  - have statically typed object models
  - compile stubs from their IDLs
  - support on-demand activation