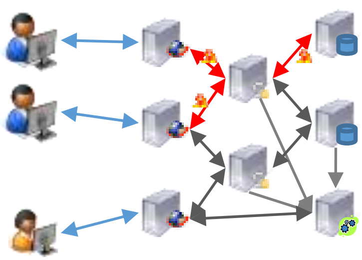


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

3

Remote Procedure Call: Basics

What is Remote Procedure Call (RPC)?

- RPC is a mechanism which allows a procedure / function F to be executed …
  - In a different address space, typically another computer on a shared network
  - And F is coded (i.e. used in code) as if it were a normal (local) procedure call
    - I.e. we have essentially the same code a local, or remote F

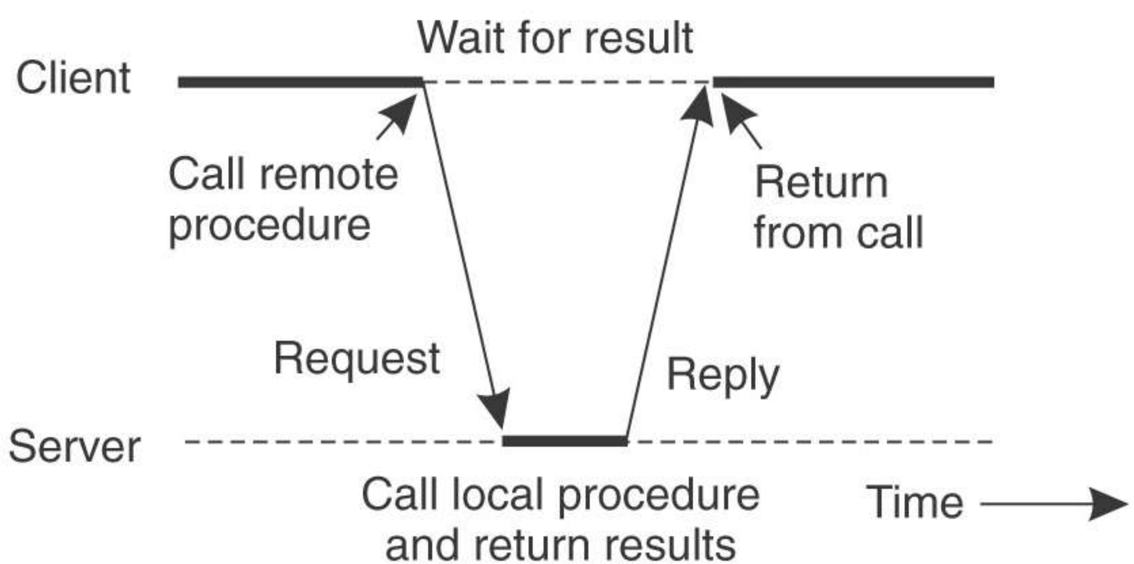
- Example: which call is remote?
  - …
  - r1 = call_A(a, b, c);
  - r2 = call_B(b, r1);
  - ..

Remote call – executed in another process or machine

Note: same syntax as for the local call!

RPC Implementation - Overview

- Idea: Server process exports an interface of functions that can be called by client programs
  - Similar to library API, class definitions, etc.
- Clients make (seemingly) local function calls
  - Under the covers, function call is converted into a message exchange with remote server process



RPC Implementation - Stubs

- A **client-side stub** is a function that looks to the client as if it were a callable function
  - I.e., same API as the “real” implementation of the function
- A **service-side stub** looks like a caller to the service
  - I.e., like a hunk of code invoking the service function

- The client program thinks it’s invoking the service
  - But it’s calling into the client-side stub
- The service program thinks it’s called by the client
  - But it’s really called by the service-side stub
- The stubs send messages to each other to make the RPC happen transparently (almost!)

RPC Implementation - Interaction

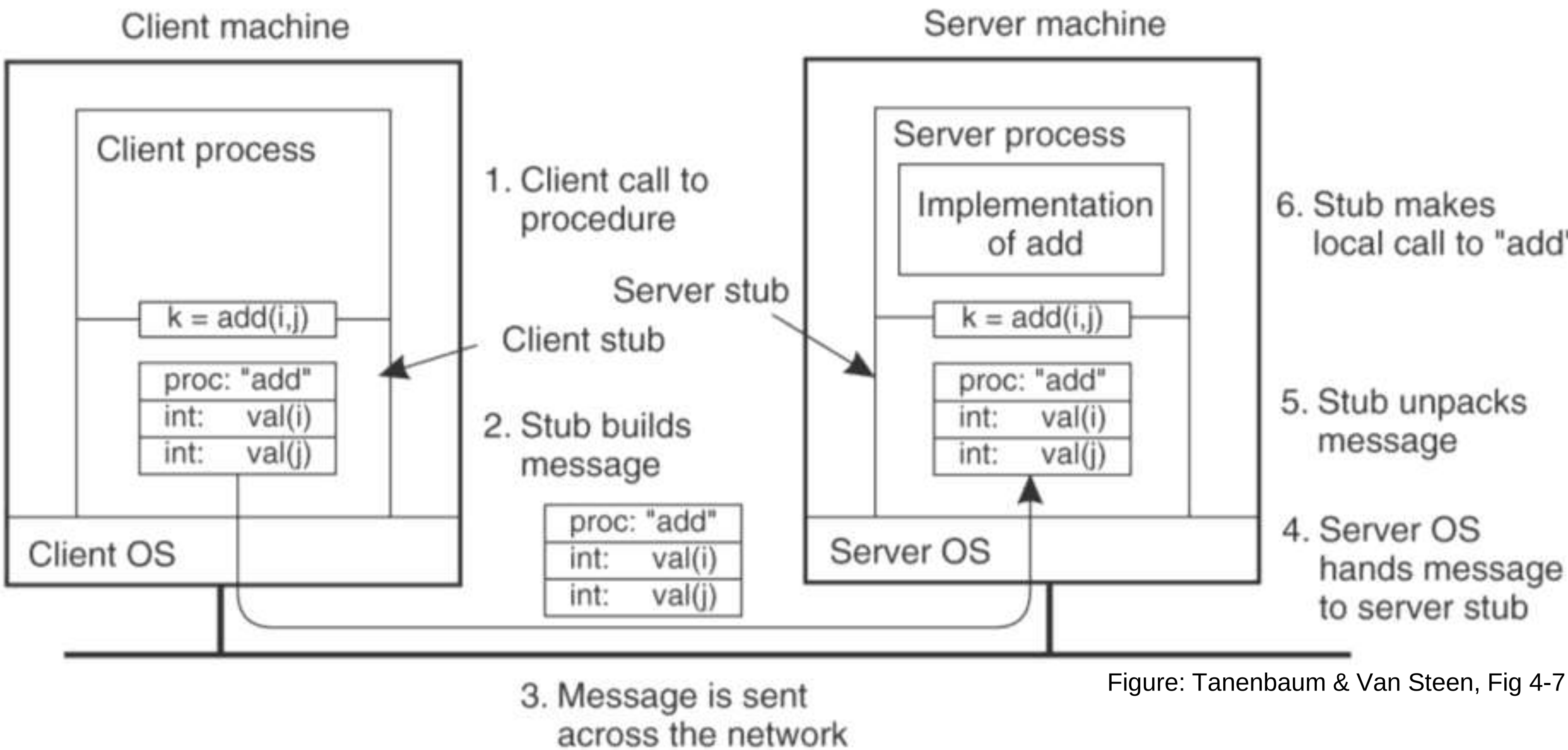


Figure: Tanenbaum & Van Steen, Fig 4-7

Stubs manage all of the details of remote communication between client and server!

RPC Stubs - Properties

- Client-side stub
  - Looks like local server function
  - Same interface as local function
  - Bundles arguments into a message, sends to server-side stub
  - Waits for reply, un-bundles results
  - Returns
- Server-side stub
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

RPC Challenges

- How to make the “remote” part of RPC invisible to the programmer?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- How to handle heterogeneity?
  - OS, language, architecture, …
- How to make it go fast?

RPC Programming and Implementation

- Server' programmer defines the service interface using an **interface definition language** (**IDL**)
  - IDL specifies the names, parameters, and types for all client-callable server procedures
- A **stub compiler** (tool) reads IDL declarations and produces <u>two</u> **stub functions** for each server function
  - Server-side and client-side
- Linking:
  - Server programmer implements the service's functions and links with the server-side stubs
  - Client programmer implements the client program and links it with client-side stubs

Marshalling Arguments

- **Marshalling** is the packing of function parameters (and return values) into a message packet
- RPC stubs call type-specific functions to marshal or unmarshal the parameters of an RPC
  - Client stub marshals the arguments into a message
  - Server stub unmarshals the arguments and uses them to invoke the service function
- On return:
  - The server stub marshals return values
  - The client stub unmarshals return values, and returns to the client program

Marshalling - Representation of Data

- **Big endian** number (Motorola, SPARK)
  - Ordered from the most significant byte (MSB) in low memory address to the least significant byte (LSB) in high memory address
- Vs. **little endian** number (Intel)
  - Ordered from the MSB in low memory address to the LSB in high memory address
- The order of the bits inside the byte is not changed
- Quiz: 0x0D0C0B0A will be represented in memory?

Little Endian

Address offset	Data
0	0A
1	0B
2	0C
3	0D

Big Endian

Address offset	Data
0	0D
1	0C
2	0B
3	0A

RPC Issue: Pointers and References /1

`read(int fd, char* buf, int nbytes)`

- Pointers are only valid within one address space
- Cannot be interpreted by another process
  - Even on same machine!

- Solution: Restricted Semantics
- Option A: **call by value**
  - Sending stub dereferences pointer, copies result to message
  - Receiving stub conjures up a new pointer
- Option B: **call by result**
  - Sending stub provides buffer, called function puts data into it
  - Receiving stub copies data to caller's buffer as specified by pointer

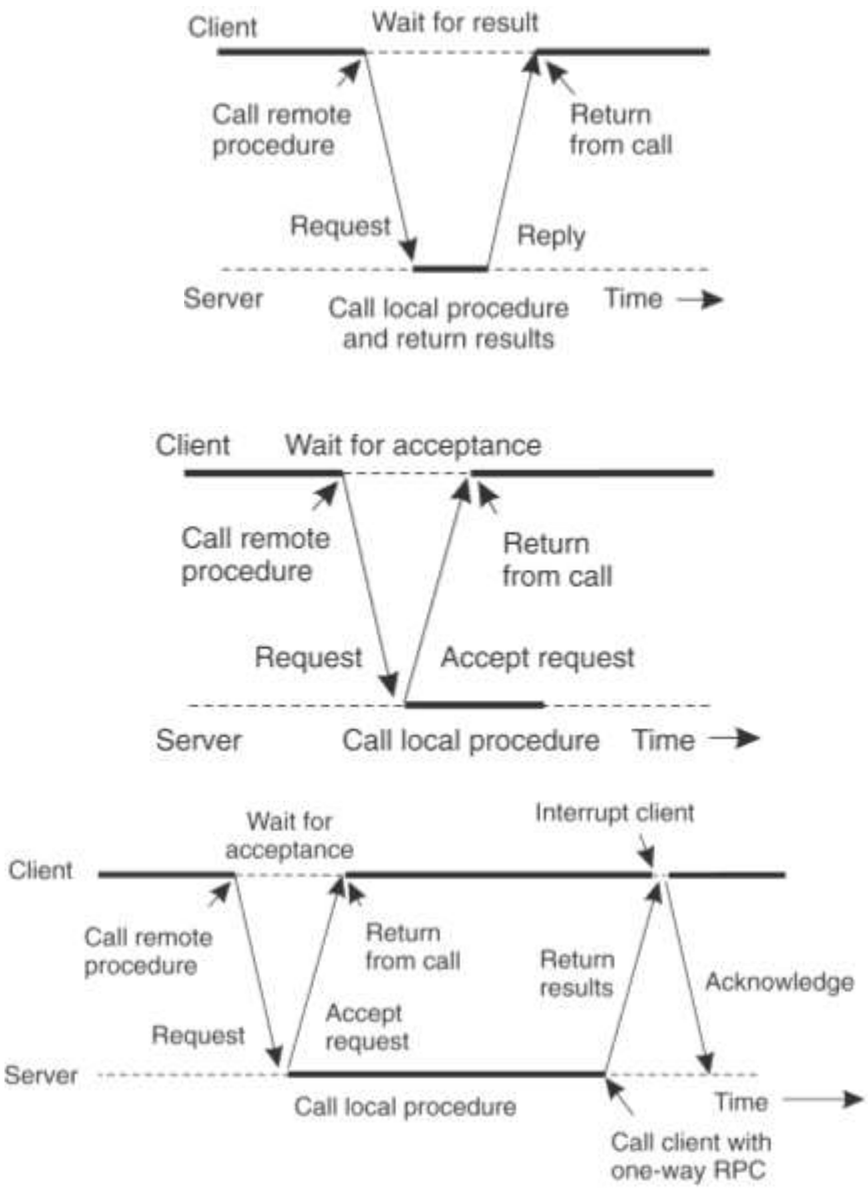
RPC Issue: Pointers and References /2

- Solution: Restricted Semantics (continued)

- Option C: **call by value-result**
  - Caller’s stub copies data to message, then copies result back to client buffer
  - Server stub keeps data in own buffer, server updates it; server sends data back in reply

- Not allowed are:
  - Call by reference
  - Aliased arguments

Synchronous vs. Asynchronous RPC



- **Synchronous** call
  - Client thread suspended until replay is received
- **Asynchronous** call
  - Client does not wait for results
  - And can actively check for them ..
  - .. Or is interrupted when results are ready

RPC Binding

- Binding is the process of connecting the client to the server
  - The server, when it starts up, exports its interface
    - Identifies itself to a network name server
    - Tells RPC runtime that it is alive and ready to accept calls
  - The client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

Practical RPC Systems

- DCE (Distributed Computing Environment)
  - Open Software Foundation; basis for Microsoft DCOM
  - Discussed in the additional slides
- Sun's ONC (Open Network Computing)
  - Widely used, very similar to DCE
  - Discussed in the additional slides
- CORBA (Common Object Request Broker Architecture)
  - Multi-language, multi-platform middleware, obj.-oriented
  - Heavyweight, "old & ugly", now very rarely used
- Java RMI (Remote Method Invocation)
  - Java-oriented approach — objects and methods
  - We will discuss in the following

RPC Example Framework: Python RpyC

RPyC: Remote Python Call

- Python library for remote procedure calls
  - RPyC: pronounced “are-pie-see”
  - https://rpyc.readthedocs.io/en/latest/
- Features (selected):
  - Transparent via object-proxying and naming conventions
  - Symmetric: both client and server can serve requests
  - Synchronous and asynchronous operation
  - Platform agnostic - 32/64 bit, little/big endian, ..., all ok
  - Low overhead due to binary protocol
- Drawbacks:
  - Only Python
  - Code maintenance might suffer (no IDL)

Example Application: Computation of Primes

```
def primes (lowerLimit, upperLimit):
primes = []
    for possiblePrime in range(lowerLimit, upperLimit + 1):
        # Assume number is prime until shown it is not
        isPrime = True
        for num in range(2, int(possiblePrime ** 0.5) + 1):
            if possiblePrime % num == 0:
                isPrime = False
                break
        if isPrime:
            primes.append(possiblePrime)

    return(primes)
```

Modified version from: https://hackernoon.com/prime-numbers-using-python-824ff4b3ea19

Server Code: Provides a “Service”

```
import rpyc
class MyService( rpyc.Service ):
    def on_connect(self, conn):       # runs when a connection is created
        print ("\n>>> RPyC service connected ...")

    def on_disconnect(self, conn):   # runs after the connection has closed
        print ("<<< RPyC service disconnected ...")

    def exposed_get_primes(self, lowerLimit, upperLimit):  # exposed method
        print ("   Starting computation on server .. ")
        list_of_primes = primes (lowerLimit, upperLimit)
        print ("   Computing finished.“)
        return list_of_primes

if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    t = ThreadedServer (MyService, port=18861)
    t.start()
```

Client Code

```
if __name__ == "__main__":
    import rpyc
    # Parameters here: server-name/IP-adr, server-port
    c = rpyc.connect("localhost", 18861)

    lower, upper = 1000000, 3*1000000
    print ("\n### Starting request ...")

    list_of_primes = c.root.get_primes(lower, upper)

    print ("### Received {} primes".format(len(list_of_primes))
```

Demo

- Scenario 1: both client and server on this laptop
- Scenario 2:
  - Client: this laptop
    - Intel Core I7-5500U @ 2.4 GHz
  - Server: linux server
    - Intel E5-1660 @ 3.3GHZ (6 cores + HT), 64 GB Ram

- What is the impact (time, share) of communication?
  - Scenario 1, Scenario 2?

RPC Example Framework: Java RMI

Java RMI

- Java language had no mechanism for invoking remote methods
  - 1995: Sun added extension
- Remote Method Invocation (RMI)
  - Allow programmer to create distributed applications where methods of remote objects can be invoked from other JVMs
- Bad
  - No goal of OS interoperability (as CORBA)
  - No language interoperability
  - No architecture interoperability
- Good
  - No need for external data representation
  - All sides run a JVM
  - Benefit: simple and clean design

RMI components

Client

- Invokes method on remote object

Server

- Process that owns the remote object

Object registry

- Name server that relates objects with names
- Similar to Internet' DNS

Needs special classes

- **Remote class** – needed for remote objects
  - Class whose instances can be used remotely
  - Within its address space: regular object
  - Other address spaces: can be referenced with an **object handle**
- **Serializable class** – needed for parameters
  - Object that can be marshaled
  - If object is passed as parameter or return value of a remote method invocation, the value will be copied from one address space to another
    - If remote object is passed, only the object handle is copied between address spaces

Additional language elements

- Stubs
  - Earlier: generated by separate compiler - **rmic**
  - No longer necessary since Java 1.5 – javac does this!
- Object registry
  - Need a remote object reference to perform remote object invocations
  - Analogous to DNS for resolving host addresses
  - Object registry does this: a demon which runs in the background
    - on linux: rmiregistry &
    - on windows: start rmiregistry

Object Registry – Server and Client

- Server
  - Register object (s) with Object Registry
    - MyClass obj = new MyClass ();
    - Naming.bind("MyClass", obj);
- Client
  - Contact **rmiregistry** to lookup name
    - MyInterface test = (MyInterface) Naming.lookup("rmi://pvs.ini.uni-heidelberg.de/MyClass");
  - **rmiregistry** returns a remote object reference
  - lookup gives reference to local stub
  - Invoke remote method(s):

    test.func("Das", "ist", "gut!");

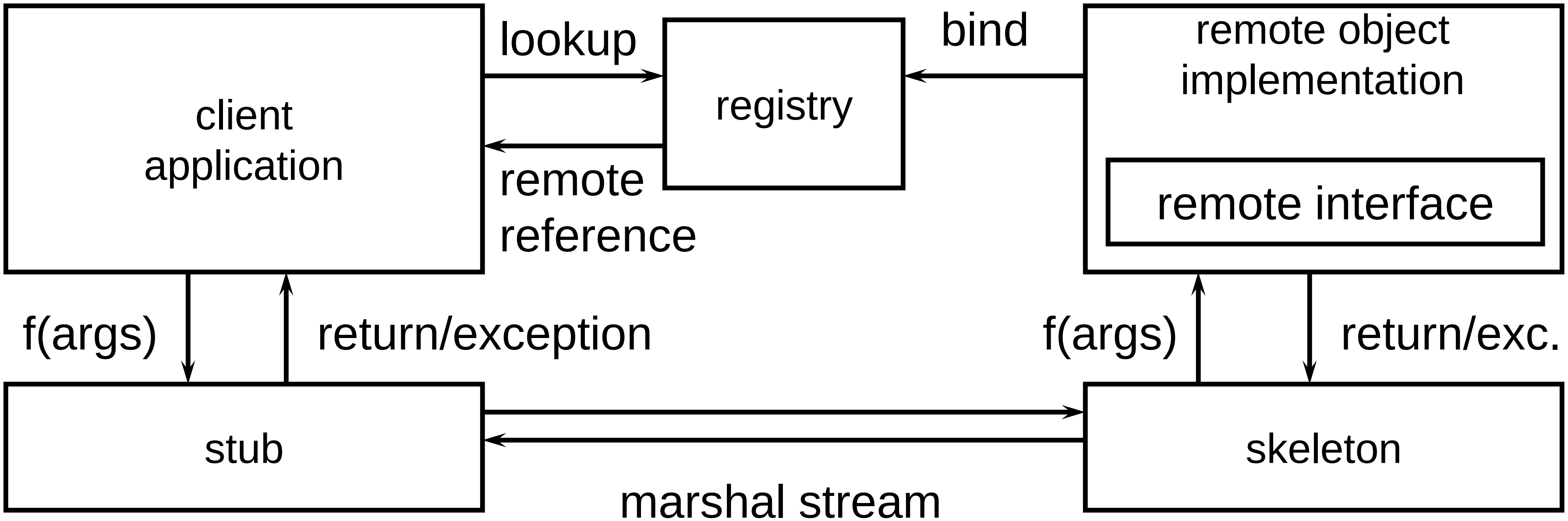
The Naming class of Java RMIregistry

- *void rebind (String name, Remote obj)*
  - This method is used by a server to register the identifier of a remote object by name
- *void bind (String name, Remote obj)*
  - This method can alternatively be used by a server to register a remote object by name,
  - but if the name is already bound to a remote object reference an exception is thrown
- *void unbind (String name, Remote obj)*
  - This method removes a binding

The Naming class of Java RMIregistry

- *Remote lookup (String name)*
  - This method is used by clients to look up a remote object by name. A remote object reference is returned
- *String [] list()*
  - This method returns an array of Strings containing the names bound in the registry

Java RMI infrastructure



Tutorials:
http://download.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html
http://littletutorials.com/2008/07/14/the-10-minutes-getting-started-with-rmi-tutorial/

Example from littletutorials.com

- Server implements an **accumulator**
  - A variable to which the value of the parameter is added and new result stored

```
class Accumulator {
    private int counter = 0;
    int incrementCounter (int value)  {
            counter += value;
            return (counter); }
}
```

- Client calls the method incrementCounter 100 times remotely, each time with parameter = 1

Interface for Remote Method

```
package com.littletutorials.rmi.api;
import java.rmi.*;

public interface Api extends Remote {
   public Data incrementCounter (Data value)
         throws RemoteException;
}
```

Serializable Class for Transmitting Data

```
package com.littletutorials.rmi.api;
import java.io.*;

public class Data implements Serializable {
    private static final long serialVersionUID = 1L;
    private int value;

    public Data(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    } }
```

Implementation of the Remote Method

```
package com.littletutorials.rmi.server;
import java.rmi.*; import java.rmi.server.*;  import com.littletutorials.rmi.api.*;

public class ApiImpl extends UnicastRemoteObject implements Api {
    private static final long serialVersionUID = 1L;
    private int counter = 0;

    public ApiImpl() throws RemoteException {
        super();
    }
    @Override
    public synchronized Data incrementCounter(Data value)
                        throws RemoteException {
        counter += value.getValue();
        return new Data(counter);
    }
}
```

Server (1): Methods to Handle Registry

```
package com.littletutorials.rmi.server;
import java.rmi.*;  import java.rmi.registry.*;  import com.littletutorials.rmi.api.*;
public class Server {
    private static final int PORT = 1099;
    private static Registry registry;

    public static void startRegistry() throws RemoteException {
        // create in server registry
        registry = java.rmi.registry.LocateRegistry.createRegistry (PORT);
    }
    public static void registerObject(String name, Remote remoteObj)
        throws RemoteException, AlreadyBoundException {
        registry.bind (name, remoteObj);
        System.out.println("Registered: " + name + " -> " +
            remoteObj.getClass().getName() + "[" + remoteObj + "]");
    }
```

Server (2): Instantiate & Register ApiImpl

```
    public static void main(String[] args) throws Exception {
        startRegistry();
        registerObject (Api.class.getSimpleName(), new ApiImpl());
        Thread.sleep (5 * 60 * 1000);
    }
}
```

Client: Get Reference from Registry + Call

```
package com.littletutorials.rmi.client;
import java.rmi.registry.*; import com.littletutorials.rmi.api.*;
public class Client {
    private static final String HOST = "localhost";
    private static final int PORT = 1099;
    private static Registry registry;

    public static void main(String[] args) throws Exception {
        registry = LocateRegistry.getRegistry (HOST, PORT);
        Api remoteApi = (Api) registry.lookup (Api.class.getSimpleName());
        for (int i = 1; i <= 100; i++) {
            System.out.println("counter = " +
                remoteApi.incrementCounter (new Data(1)).getValue());
            Thread.sleep(100);
} } }
```

Thank you.

Additional Slides:
RPC Example Framework -
SUN’s Open Network Computing

RPC under Unix / Linux

- One of the first RPC systems was described in 1976 (Request For Comments (RFC) 707)
- The first popular implementations was **SUN‘s RPC**
  - Now known as **ONC RPC** (**Open Network Computing**)
  - described in RFC 1831 (published in 1995)
  - RFC 5531, published in 2009, is the current version
- Implementations exist for
  - Unix System V, Linux, BSD, OS X
  - Microsoft Windows Services for UNIX
  - Third-party for C/C++, Java, .NET

The ONC RPC Architecture

<u>Server</u>

- Each **RPC-service** (local program which responds to RPC requests) tells a special local demon "port mapper" (**portmap** or **rpcbind**) - on which port it listens for requests
- The port mapper acts as a "directory"/ "telephone book" and tells clients the ports of the "final" service
  - portmap listens on port 111

<u>Client</u>

- Client needs to contact the port mapper first
- It submits a RPC program number
- … and receives the port number x of the targeted program (RPC-service)
- Then it makes a RPC-call to this program via port x

Program Numbers Specify Programs

- See [IANA specification]
- Example:

Description/Owner	RPC Prg Number	Short Name
portmapper	100000	pmapprog portmap rpcbind
remote stats	100001	rstatprog
remote users	100002	rusersprog
nfs	100003	nfs
yellow pages (NIS)	100004	ypprog ypserv
mount demon	100005	mountprog
…		

Mapping PRG# to Service Ports (Example)

- $ rpcinfo -p localhost
- program vers proto port
- 100000 2 tcp 111 portmapper
- 100000 2 udp 111 portmapper
- 100003 2 udp 2049 nfs
- 100003 3 udp 2049 nfs
- 100003 4 udp 2049 nfs
- 100003 2 tcp 2049 nfs
- 100003 3 tcp 2049 nfs
- 100003 4 tcp 2049 nfs
- 100024 1 udp 32770 status
- 100024 1 tcp 32769 status
- 100021 4 tcp 32769 nlockmgr
- 100005 1 udp 644 mountd
- 100005 1 tcp 645 mountd

Implementing ONC RPC

- Three steps
  - Specify the protocol for client server communication
  - Develop the server program
  - Develop the client program
- Materials:
  - Tutorial: Chapter 33 of A. D. Marshall, Programming in C - UNIX System Calls and Subroutines using C, link
  - rpcgen Programming Guide, link
  - ONC+ RPC Developer's Guide: link

Specifing the Protocol

- Developer specifies name of the service procedures, data types of parameters and return arguments
- Done via language called **external data representation** (**XDR**)
  - At first XDR only for machine-independent communication
  - Later extended to be an **Interface Definition Language** - **IDL**
- From this data (file) the program rpcgen generates:
  - *_clnt.c -- the client stub; *_svc.c -- the server stub;
  - *_xdr.c -- If necessary XDR filters (marshalling and unmarshalling procedures)
  - *.h -- the header file needed for any XDR filters

SUN XDR Example

- /** FileReadWrite service interface
- definition in file FileReadWrite.x */
- const MAX = 1000;
- typedef int FileIdentifier;
- typedef int FilePointer;
- typedef int Length;
- struct Data {
  - int length;
  - char buffer[MAX];
- };
- struct writeargs {
  - FileIdentifier f;
  - FilePointer position;
  - Data data;
- };
- struct writeargs {
- FileIdentifier f;
- FilePointer position;
- Data data;
- };
- struct readargs {
- FileIdentifier f;
- FilePointer position;
- Length length;
- };
- program FILEREADWRITE {
- version VERSION {
- void WRITE(writeargs)=1;
- Data READ(readargs)=2;
- }=2;
- } = 9999;

Simplified Interface

- For most applications, it is sufficient to use rpcgen and know the simplified interface (http://goo.gl/Vw40S)
- **rpc_reg**()
  - Registers a procedure as an RPC program
- **rpc_call**()
  - Calls the specified procedure on the specified remote host
- **rpc_broadcast**()
  - Calls a specified procedure on all hosts reachable via a UDP-broadcast (link)

Example – rpc_call

- int rpc_call (
  - char *host /* Name of server host */,
  - u_long prognum /* Server program number */,
  - u_long versnum /* Server version number */,
  - xdrproc_t inproc /* XDR filter to encode arg */,
  - char *in /* Pointer to argument */,
  - xdr_proc_t outproc /* Filter to decode result */,
  - char *out /* Address to store result */,
  - char *nettype /* For transport selection */);

Sun RPC – Advantages

- Don’t worry about getting a unique transport address (port)
  - Applications need to know only one transport address – i.e. 111 of port mapper
  - But you need a unique program number per server
- Transport independent
  - Protocol can be selected at run-time
- Application does not have to deal with maintaining message boundaries, fragmentation, reassembly
- Function call model can be used instead of send/receive

Additional Slides (RPC System): DCE - Distributed Computing Environment

DCE RPC

- **DCE -** Distributed Computing Environment
  - A software system developed in the early 1990s by Open Software Foundation (OSF)
    - Consortium of Apollo Computer (later part of HP), IBM, DEC, …
- A framework for developing client/server applications
- Provided several components
  - **DCE/RPC** - a remote procedure call mechanism
  - A naming (directory) service
  - A time service
  - An authentication service
  - **DCE/DFS** - a distributed file system (DFS)
- More on this in additional slides …

DCE/RPC

- an example of design by committee – too complex
- Interfaces written in a language called **Interface Definition Notation** (**IDN**)
  - Definitions look like function prototypes
- Authenticated RPC support with **DCE security services**
- Integration with **DCE directory services** to locate servers
- Marshalling: standard formats for data
  - **NDR**: **Network Data Representation**
- Run-time libraries
  - One for TCP/IP and one for UDP/IP

Unique IDs

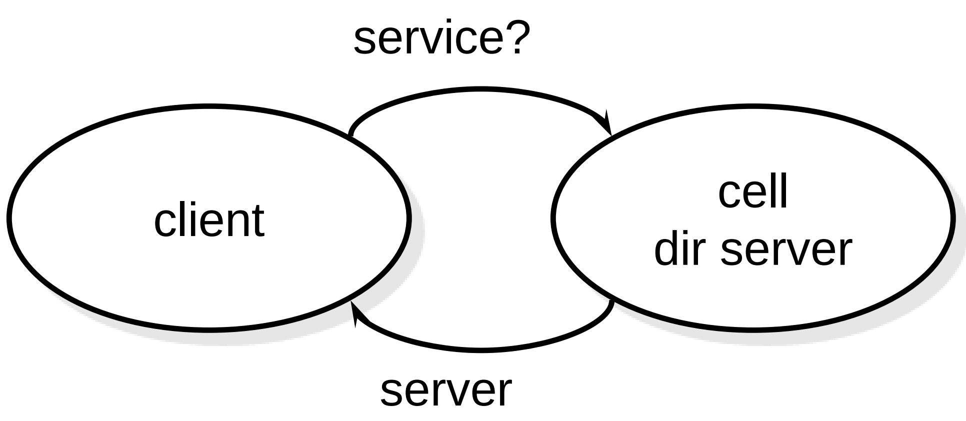
- Sun RPC required a programmer to pick a “unique” 32-bit number for his routine / program

- In DCE, program gets a unique ID with **uuidgen**
  - Generates prototype IDN file with a 128-bit Unique Universal ID (UUID)
  - 10-byte timestamp multiplexed with version number
  - 6-byte node identifier (ethernet address on ethernet systems)

Service lookup

- Sun RPC requires client to know name of server
- DCE allows several machines to be organized into an administrative entity
  - A **cell** (collection of machines, files, users)
- **Cell directory server**
  - Each machine communicates with it for cell services information

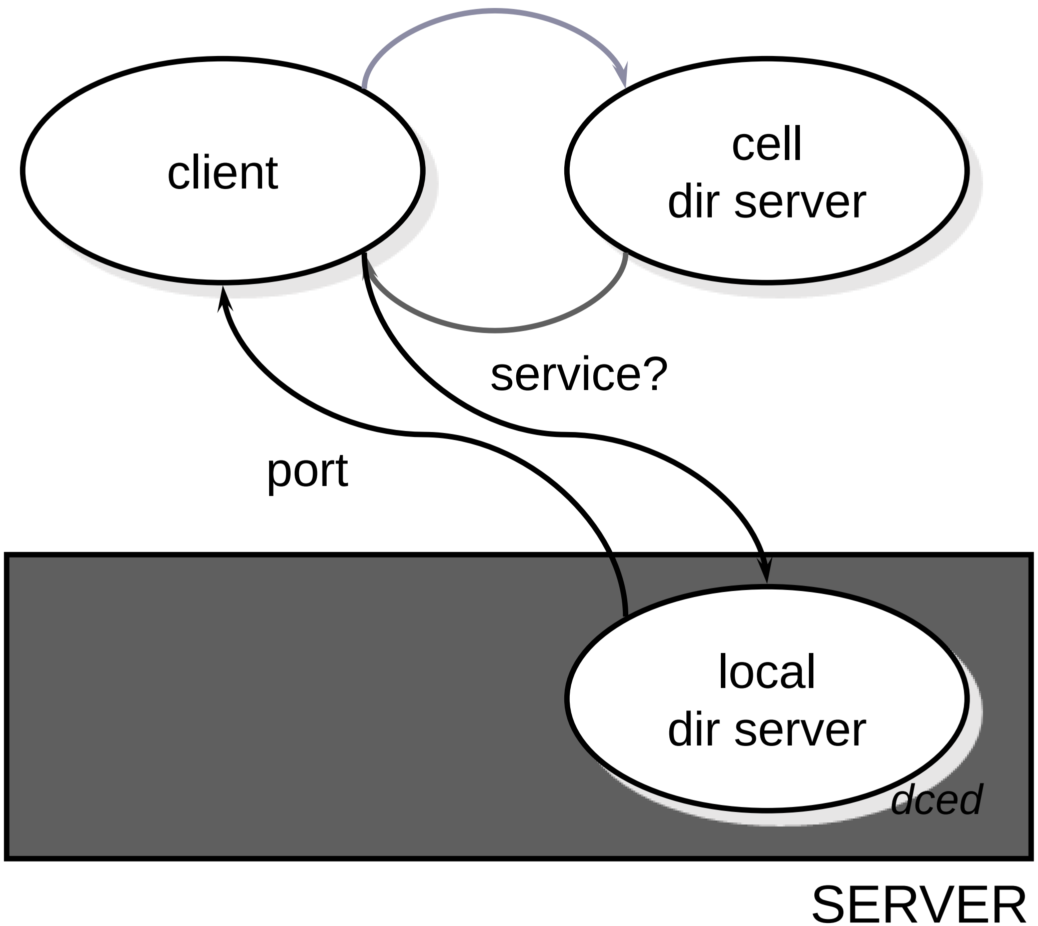
DCE service lookup



Request service lookup from cell directory server

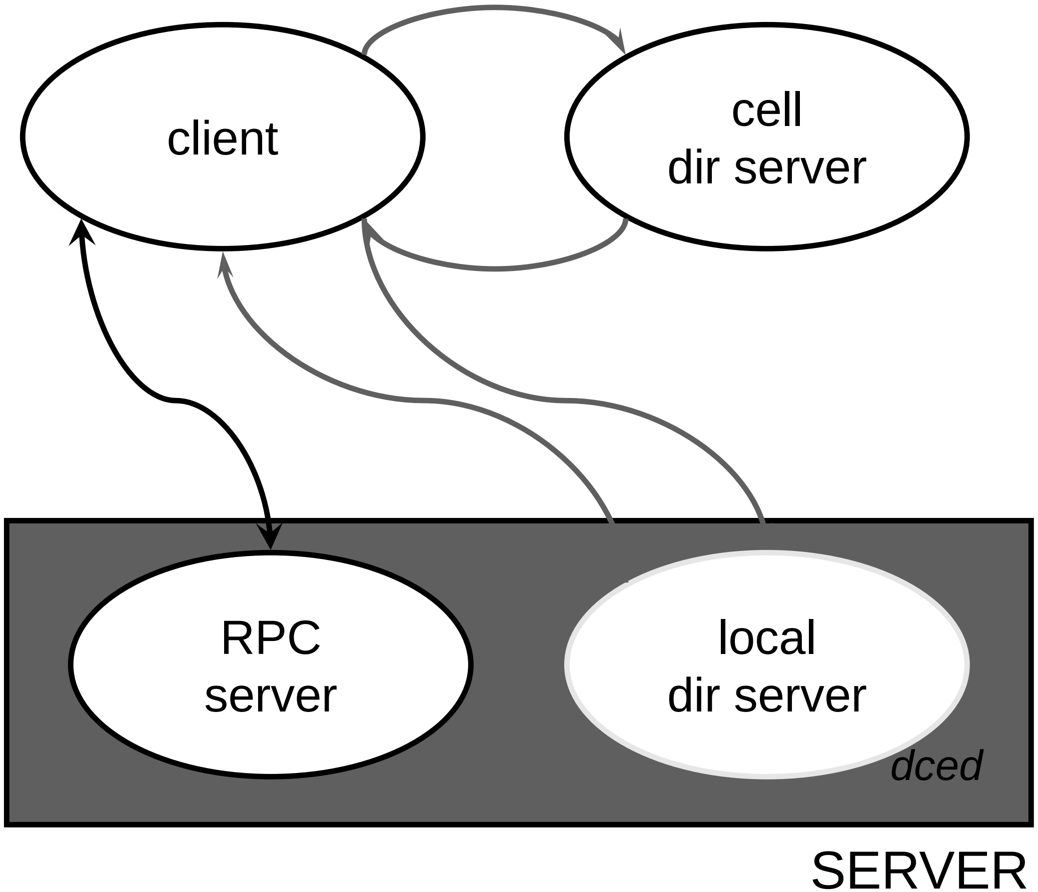
Return server machine name

DCE service lookup



Connect to endpoint mapper service and get port binding from this local name server

DCE service lookup



Connect to service and request remote procedure execution

Sun RPC and DCE RPC Disadvantages

- If server is not running:
  - Service cannot be accessed
  - Administrator is responsible for starting it

- If a new service is added:
  - There is no mechanism for a client to discover this

- Object oriented languages expect polymorphism
  - Service may behave differently based on data types passed to it
  - Not supported by both standards

Additional Slides: CORBA

CORBA

CORBA

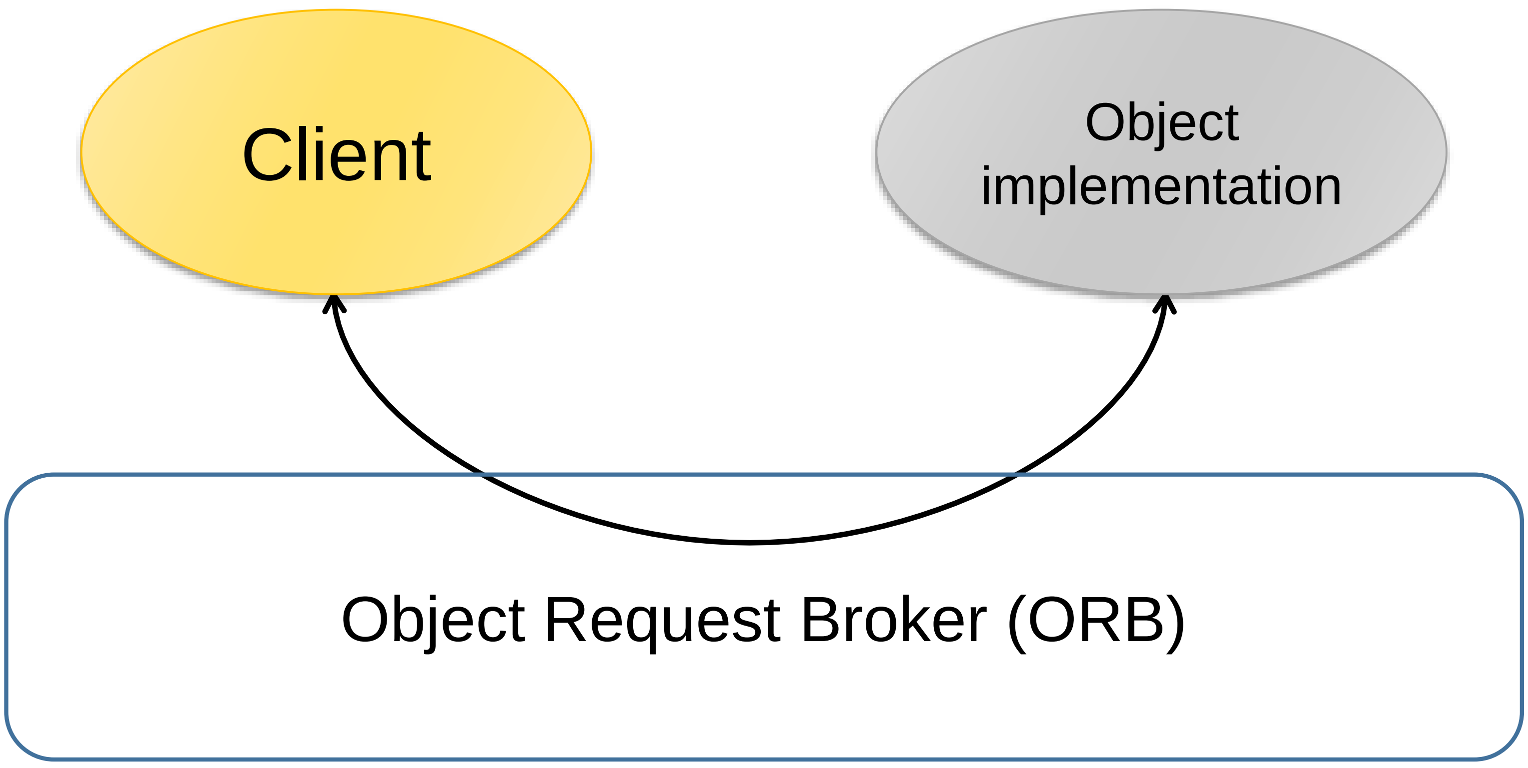
- **Common Object Request Architecture**
- A standard defined by the **Object Management Group** (**OMG**)
  - Consortium of >700 companies
- Enables software components written in <u>multiple computer languages</u> and running on <u>multiple computers</u> to work together
  - Enables “distributing objects”
  - Specification is independent of any language, OS, network
  - Version 1.0 released in 1991, still evolving

CORBA

- Basic paradigm:
  - Request services (i.e. call methods) of a distributed (i.e. remote) object
- So what are the differences to ONC RPC etc.?
  - 1. All interactions pass through a specific middleware which “unifies” communication and data formats
    - This middleware is called **Object Request Broker** (**ORB**)
  - 2. Distributed components can be written in different computing languages
  - 3. There are a lot of additional services which support development of distributed applications and systems

1. Object Request Broker (ORB)

- Logical view: ORB delivers request to the object and returns results to the client



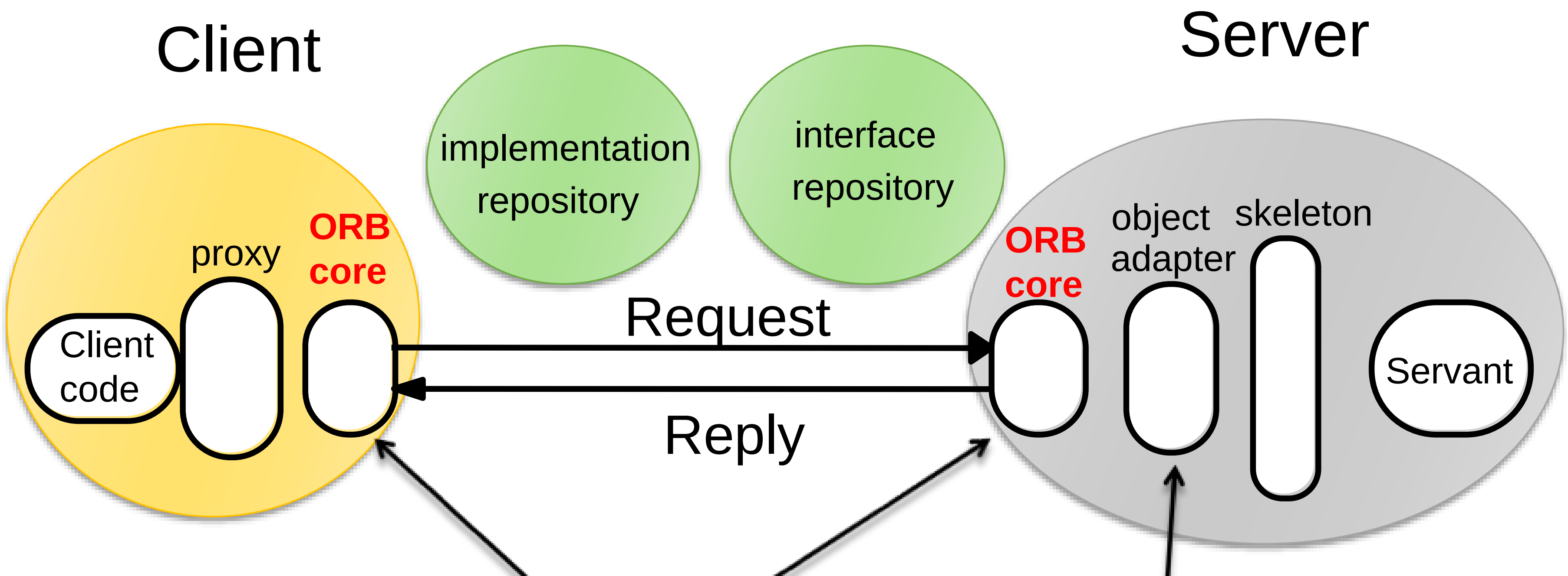
1. Object Request Broker (ORB) /2

- Distributed service that implements the request to the remote object
  - Locates the remote object on the network
  - Communicates request to the object
  - Waits for results
  - Communicates results back to the client
- Responsible for providing location transparency
  - Same request mechanism used by client & CORBA object regardless of object location (local or remote)
- Client request may be written in a different programming language than the implementation

More ORB functions

- Look up and instantiate objects on remote machines
- Marshal parameters
- Deal with security issues
- Publish data on objects for other ORBs to use
- Invoke methods on remote objects
  - Static or dynamic execution
- Automatically instantiate objects that aren't running
- Route callback methods
- Communicate with other ORBs

1. CORBA and ORB – Architectural View



- Application first initializes the ORB core
- Then it accesses an internal **Object Adapter**
  - It maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies

2. Different Programming Languages

- The idea is to have a language-neutral interfaces which describe objects to outer world
  - This is done with the an interface definition language (**IDL**)
- The second component is a **mapping** from IDL to a specific implementation language
  - This is like a „translator“, a library created specifically for each language
  - Currently standardized language bindings for: C, C++, Java, Ada, COBOL, Smalltalk, Objective C, LISP, Python

Writing Applications in CORBA

A developer needs to:

1. Write a description of the objects and interfaces in IDL
2. Compile it (with tools of a CORBA-implementation)
   - Result are **generated code classes** which translate high-level interface definition into an OS- and language-specific code for the user application
   - At run-time, these are registered with the Object Adapter

IDL (Interface Definition Language)

- IDL description of (distributed) objects is easy in some languages
  - Java, Python – due to similar “structure” of these languages
- … And very difficult in others
  - C – programmer needs to emulate OOP-features
  - C++ - programmer needs to learn “complex and confusing datatypes“ that predate C++ STL
- IDL data types
  - Basic types: long, short, string, float, …
  - Constructed types: struct, union, enum, sequence
  - Typed object references
  - The any type: a dynamically typed value

IDL example - Java

```
struct Rectangle {
    long width;
    long height;
    long x;
    long y;
} ;

struct GraphicalObject {
    string type;
    Rectangle enclosing;
    boolean isFilled;
};

interface Shape {
    long getVersion();
     // returns state    of the GraphicalObject
    GraphicalObject getAllState();
};
```

IDL example - Java

```
typedef sequence <Shape, 100> All;

interface ShapeList {
        exception FullException{ };
        Shape newShape (in GraphicalObject g)
                raises(FullException);

        // returns sequence of remote object references
        All allShapes();
        long getVersion() ;
};
```

IDL Constructed Types – 1 of 2

Type	*Examples*	*Use*
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All* bounded and unbounded sequences of *Shapes*	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*String name;* *typedef string<8> SmallString;* unbounded and bounded sequences of characters	Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.
array	*typedef octet uniqueId[12];* *typedef GraphicalObject GO[10][8]*	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.

IDL Constructed Types – 2 of 2

Type	*Examples*	*Use*
record	*struct GraphicalObject {* *string type;* *Rectangle enclosing;* *boolean isFilled;* *};*	Defines a type for a record containing a group of related entities. *Structs* are passed by value in arguments and results.
enumerated	*enum Rand* *(Exp, Number, Name);*	The enumerated type in IDL maps a type name onto a small set of integer values.
union	*union Exp switch (Rand) {* *case Exp: string vote;* *case Number: long n;* *case Name: string s;* *};*	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an *enum*, which specifies which member is in use.

Objects

- Object references <u>persist</u>
  - They can be saved as a string
  - … and be recreated from a string
- **Client**
  - Performs requests by having an **object reference** for object and desired operation (method)
  - Client initiates request by
    - Calling stub routines specific to an object
    - Or constructing request dynamically (**DII interface**)
- **Server** (object implementation)
  - Provides semantics of objects
  - Defines data for instance, code for methods

3. CORBA Services (**COS**)

- CORBA provides a set of distributed services to support programming of distributed CORBA-apps
  - Standard CORBA objects with IDL interfaces
- Popular services
  - Object life cycle
    - Defines how CORBA objects are created, moved, removed, copied
  - Naming
    - Defines how objects can have friendly symbolic names
  - Events
    - Asynchronous communication
  - Externalization
    - Coordinates the transformation of objects to/from external media (in Java: serializing)

Even More Popular services

- Transactions
  - Provides atomic access to objects
- Concurrency control
  - Locking service for serializable access
- Property
  - Manage name-value pair namespace
- Trader
  - Find objects based on properties and describing service offered by object
- Query
  - Queries on objects

Interoperability

- CORBA clients are portable
  - They conform to the API … but may need recompilation
- Object implementations (servers)
  - generally need some rework to move from one vendor's CORBA product to another
- CORBA 2.0 defined network protocol called **IIOP**: **Inter-ORB Protocol** (1996)
  - IIOP works across any TCP/IP implementations
- IIOP can be used in systems that do not even provide a CORBA API
  - Used as transport layer in some Java RMI versions
    - **RMI over IIOP**

CORBA vendors

- Lots of vendors
  - ORBit
    - Bindings for C, Perl, C++, Lisp, Pascal, Python, Ruby, and TCL
  - Java ORB
    - Part of Java SDK
  - VisiBroker for Java
    - From Imprise; embedded in Netscape Communicator
  - OrbixWeb
    - From Iona Technologies
  - Websphere
    - From IBM
  - Many others

Assessment

- Reliable, comprehensive and language neutral
- Standardized and supported by many vendors
- But (more criticism here)
  - Due to “design by committee“ flaws CORBA is complex, expensive to implement and often ambiguous
  - Steep learning curve
  - Integration with languages not always straightforward
  - Supports only tightly coupled environments
    - Not very suitable for WANs / Internet