department of computer science
faculty of sciences

Internet Programming

Client-Server Programming

Introduction

Introduction

- Now that you know how to make computers communicate, you can write any distributed application!
  - Decide which machine does what
  - Identify when computers need to communicate and what they should tell each other
  - Program it!

THE END

(well, almost...)

…except that

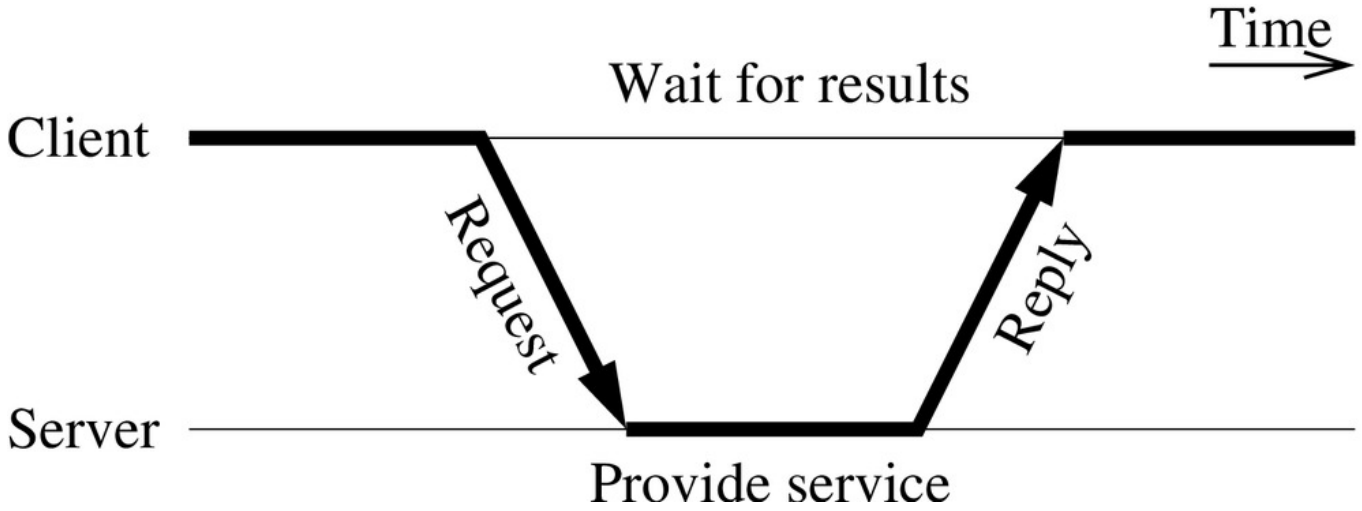
- You will have to deal with many issues:
  - How to **structure** your program
  - Define an **application protocol**
    - That is powerful enough for all your needs
    - Yet efficiently implemented
  - Deal with machines of **different architectures**
    - They may represent data differently
  - **Locate machines**
    - "Which machine implements task X?"
  - etc.

- You are in need of a middleware

- A **middleware** is a piece of software in charge of these issues
  - You program the application code
  - The middleware takes care of distribution issues (at least some of them)

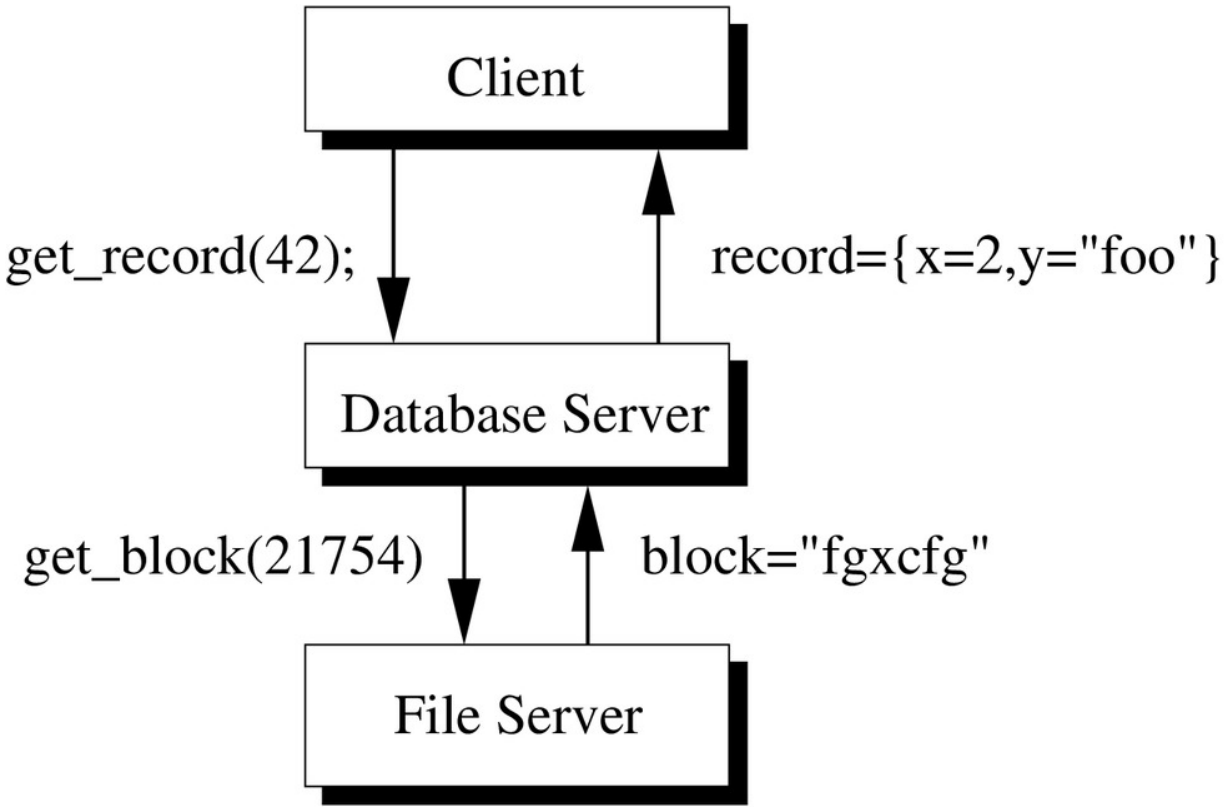
The Client-Server Model

- This is the most used model for organizing distributed applications
- **Servers** implement specific services
  - e.g., a file system, a database service
- **Clients** request services from the servers, and wait for the response before continuing



Chained Client-Server Interactions

- A server can itself be a client to another server:



- You just have to be careful about loops (A→B→C→A)

Why use the Client-Server Model?

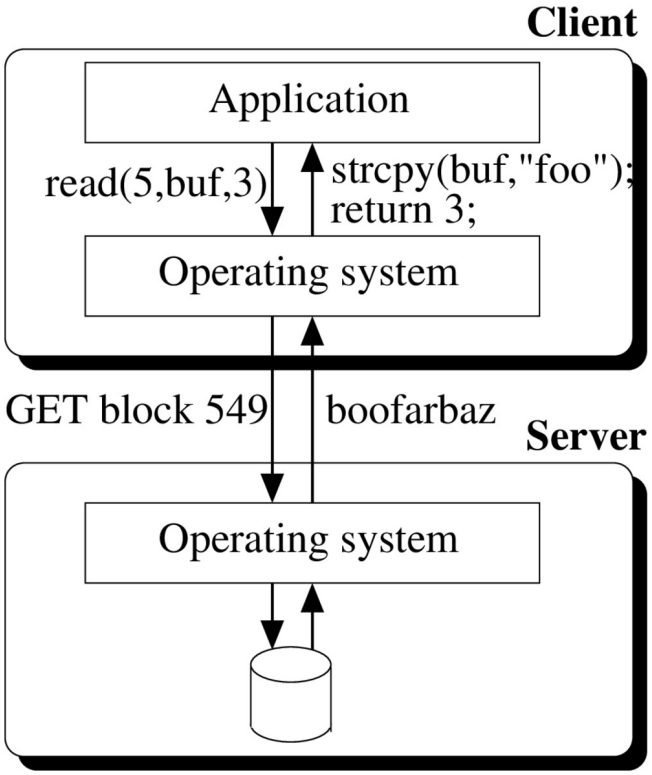
- A client-server application (usually) **does not run faster** than a centralized application
  - The client is waiting while the server works
  - Additional delays due to communication

- But it has several advantages!

- **QUESTION:** What are they? ☺
  - Benefitting from **specialized resources**
    - Fast CPU, lots of memory, etc.
    - Special device attached (printer, disk array, GPS clock, etc.)
  - **Splitting up an application**
    - If it is too big to fit in one computer (memory space, disk, etc.)
  - **Sharing information** between multiple clients
    - A file server allows file sharing between multiple clients
    - Same for a database server (data sharing), etc.

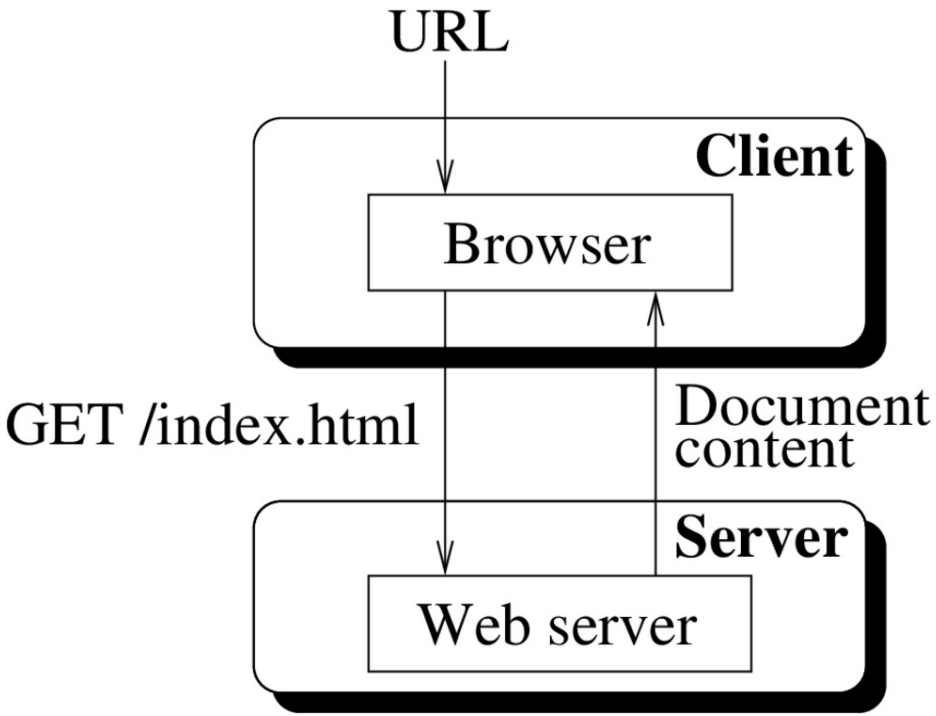
Example

- **Distributed File Systems**
  - The operating system contains specialized code
  - Convert file-related system calls to requests to the file server
  - Convert server replies into system call return values
- The whole system is built specifically for one given application



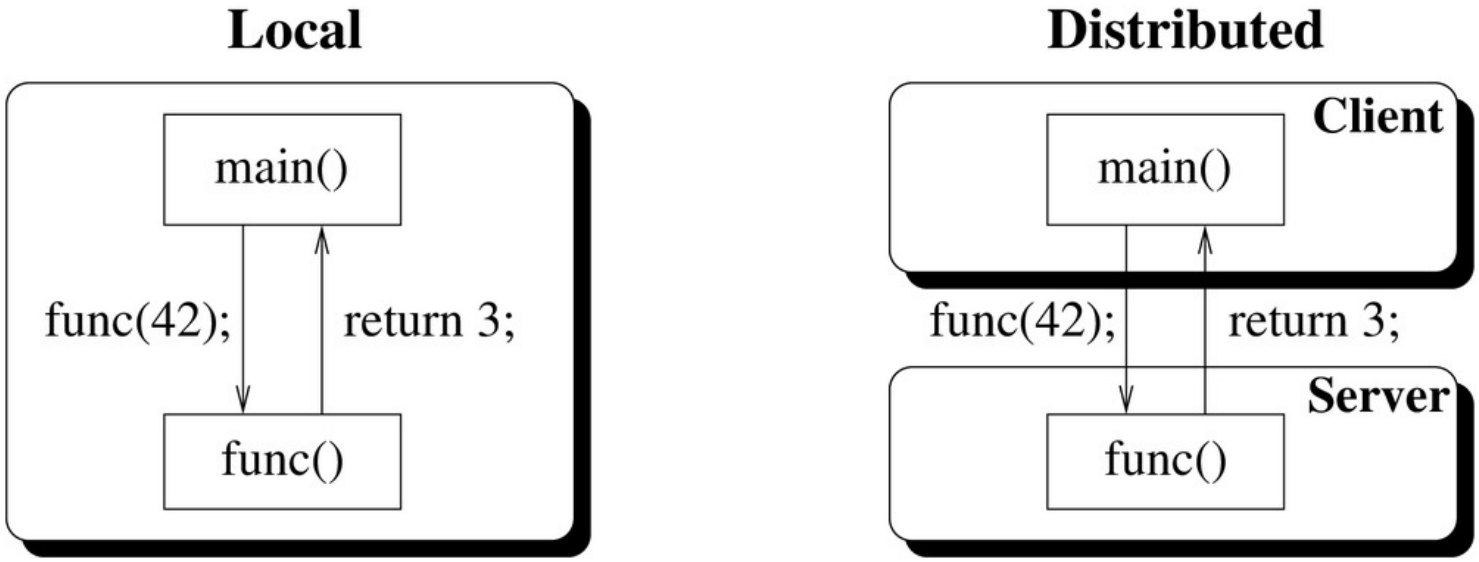
Another Example

- **The World-Wide Web**
  - A specialized client-server protocol has been defined: HTTP



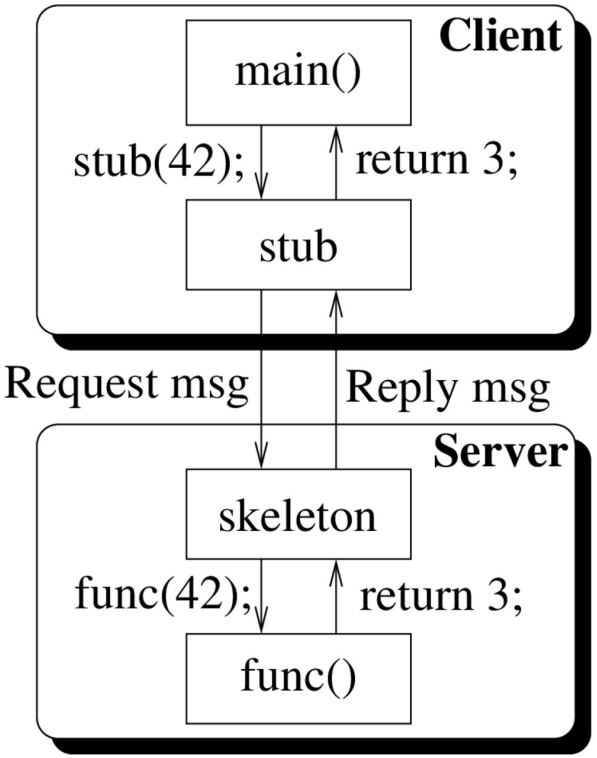
Remote Procedure Calls [1/2]

- There already exists a way to represent a task in a local application: **procedures** (or **functions**)
- Let's extend the model to **remote procedures**



Remote Procedure Calls [2/2]

- You need to convert invocations into network messages and vice-versa
  - A **stub** is a function with the same interface as `func()`: It converts function calls into network requests, and network responses into function returns
  - A **skeleton** converts network requests into function calls and function responses into network replies



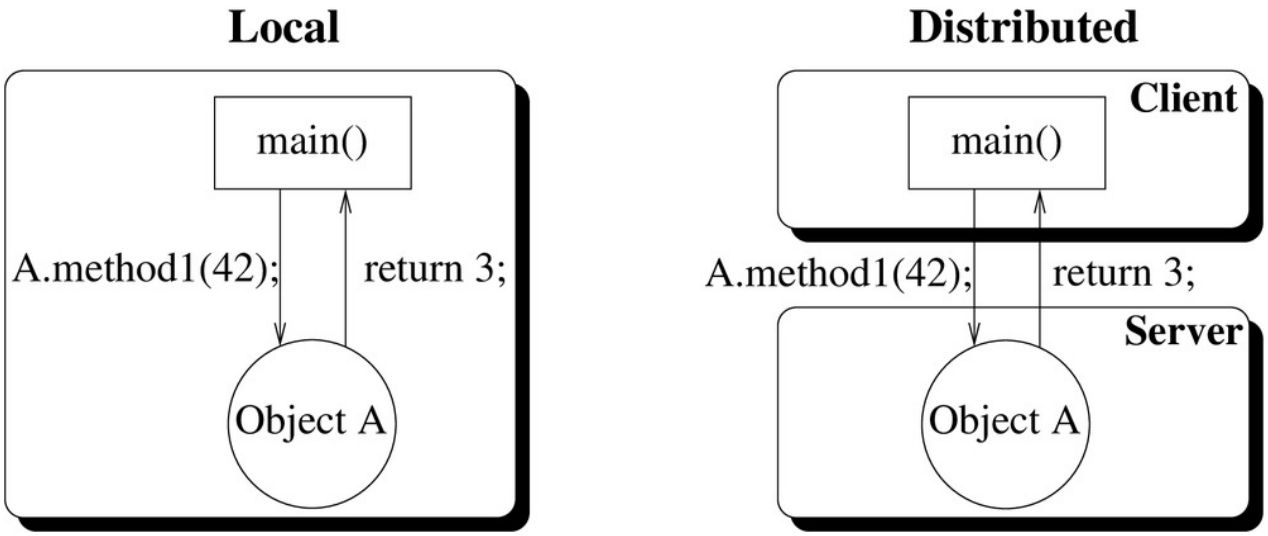
- An RPC system is used to generate the stub and the skeleton (more or less) automatically
  - Based on a description of the interface of `func()`

Limitations of RPC

- Clients and servers do not share the same **address space**
- Contrary to non-distributed programs, clients and servers:
  - Do not share **global variables**
  - Do not share **file descriptors**
    - Therefore the server cannot directly access a file opened by the client
  - Cannot use **pointers** as function parameters
    - Because the server will not be able to follow such pointers
- This sets constraints on which parts of a program you can separate and run as a server
  - The server must have a clear **interface** (a set of function prototypes)
  - The server can have **internal data**, but clients **cannot access** them
  - The client can have **internal data**, but the server **cannot access** them

Remote Method Invocation

- The equivalent to RPC in the object-oriented world is **RMI (Remote Method Invocation)**



- Like in RPC, you must have **stubs** and **skeletons**
- There are several Remote Method Invocation systems:
  - Sun RMI (entirely in Java), Corba (language independent), etc.

Using Sun RPC

Introduction

- Sun RPC is a Remote Procedure Call system
  - Officially called ONC-RPC
  - "Open Network Computing Remote Procedure Call"
- It is very widely used
  - It was originally developed by Sun Microsystems (now bought by Oracle), but is now implemented in most (all?) Unix systems
  - Also implemented on Windows
  - The NFS distributed file system is based on Sun RPC
- It is **platform independent**
  - A Linux computer can call a procedure on a Solaris box, etc.
- But **not language independent**
  - Designed to call remote **C procedures**
  - But you can also use other languages that have gateways to C

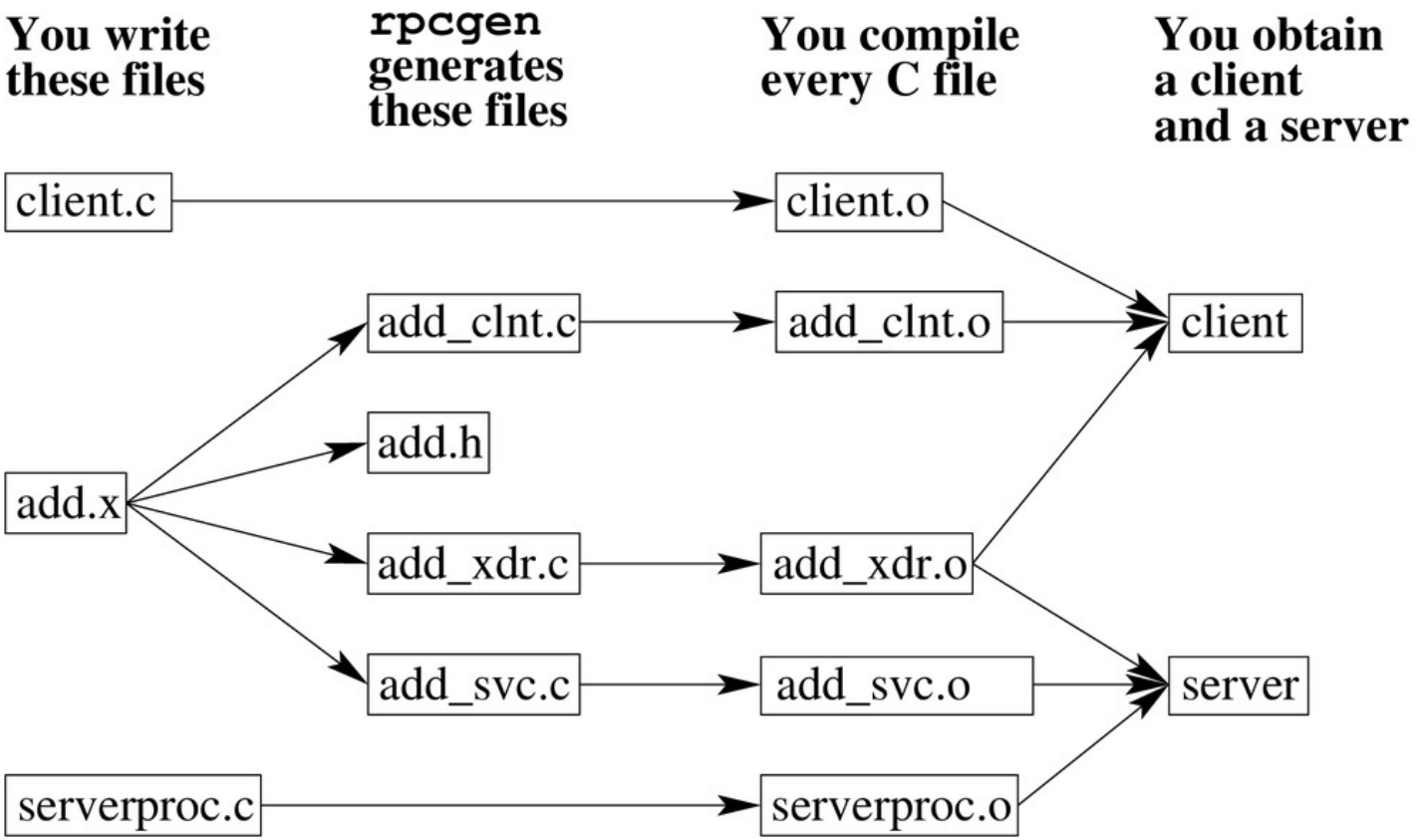
Writing an RPC Program [1/3]

- To write a minimalist RPC program, you must write:
  - A C procedure to be remotely called: **remoteproc.c**
  - A specification of the procedure: **remoteproc.x**
  - A client program that calls the procedure: **client.c**

Writing an RPC Program [2/3]

- Based on **remoteproc.x**, the program rpcgen generates:
  - A **header file** that you will include in both programs: **remoteproc.h**
  - A **client stub**, that your client can use to send an RPC: **remoteproc_clnt.c**
  - A **server skeleton**, which will call your procedure when a request is received: **remoteproc_svc.c**
  - **Internal functions** to convert the procedure parameters into network messages and vice-versa: **remoteproc_xdr.c**
  - Beware: on Solaris, rpcgen generates K&R code by default.
    - Use **rpcgen –C** to generate ANSI code

Writing an RPC Program [3/3]



A Bit of Terminology

- One computer can be a server of multiple procedures
- But how is it organized?
  - A server may host several **programs** (identified by a **program number**)
  - Each program may have several subsequent **versions** (identified by a **version number**)
  - Each version of a program may contain one or more **procedures** (identified by a **procedure number**)
- Program numbers are 32-bit hexadecimal values (e.g., 0x20000001)
  - As a user, you can choose any program number between 0x20000000 and 0x3FFFFFFF
  - **But make sure program numbers are unique!**
    - You cannot have several programs with the same number on the same machine
- **Version** and **procedure numbers** are integers (1, 2, …)

An RPC Example

- Start by writing the specification file: **add.x**

```
struct add_in {     /* The arguments of the procedure */
    long arg1;
    long arg2;
};

typedef long add_out;  /* The return value of the procedure */

program ADD_PROG {
  version ADD_VERS {
    add_out ADD_PROC(add_in) = 1; /* Procedure number = 1 */
  } = 1;                          /* Version number = 1 */
} = 0x20001234;                   /* Program number = 0x20001234 */
```

An RPC Example

- This file contains specifications of:
  - A structure **add_in** containing the arguments
  - A typedef **add_out** containing the return values
  - A program named **ADD_PROG** whose number is 0x20001234
  - The program contains one version with value **ADD_VERS** = 1
  - The version contains one procedure with value **ADD_PROC** = 1
    - This procedure takes an **add_in** as parameter, and returns an **add_out**

- Remarks
  - Your procedures can only take **one** input argument and return **one** output return value
  - If you need more arguments or return values, group them into a structure (like **add_in**)
  - Fields that are represented as pointers (e.g., char *) should not be assigned NULL! They should point to some actual content.

An RPC Example

- Generate stubs:

```
rpcgen add.x
```

- **add.h** contains various declarations:

```
#define ADD_PROG 0x20001234          /* Program nb */
#define ADD_VERS 1                   /* Version nb */
#define ADD_PROC 1                   /* Procedure nb */
add_out * add_proc_1(add_in *, CLIENT *);
add_out * add_proc_1_svc(add_in *, struct svc_req *);
```

  - **add_proc_1:** the stub, i.e., the procedure that the client program will call
  - **add_proc_1_svc:** the actual procedure that you will write and run at the server

- **add_clnt.c** contains the implementation of add_proc_1

- **add_svc.c** contains a program which calls your procedure add_proc_1_svc when it receives a request

- **add_xdr.c**: marshall/unmarshall routines

An RPC Example

- Write your server procedure: **serverproc.c**

```
#include "add.h"

add_out *add_proc_1_svc(add_in *in, struct svc_req *rqstp) {
  static add_out out;
  out = in->arg1 + in->arg2;
  return(&out);
}
```

  - **rqstp** (request pointer): contains some information about the requester
    - Its IP address, etc.

An RPC Example

- Compile your server

- You need to compile together your procedure, the (generated) server program, the (generated) marshal/unmarshal procedures and the *nsl* library
  - The nsl library contains the RPC runtime

```
$ gcc -c serverproc.c
$ gcc -c add_svc.c
$ gcc -c add_xdr.c
$ gcc -o server serverproc.o add_svc.o add_xdr.o -lnsl
```

- You can start your server:

```
./server
```

An RPC Example

- Write the client program: client.c

```
#include "add.h"
int main(int argc, char **argv) {
  CLIENT *cl;
  add_in in;
  add_out *out;

  if (argc!=4) {
    printf("Usage: client <machine> <int1> <int2>\n\n";
    return 1;
  }

  cl = clnt_create(argv[1], ADD_PROG, ADD_VERS, "tcp");
  in.arg1 = atol(argv[2]);
  in.arg2 = atol(argv[3]);
  out = add_proc_1(&in, cl);
  if (out==NULL) { printf("Error: %s\n",clnt_sperror(cl,argv[1])); }
  else { printf("We received the result: %ld\n",*out); }
  clnt_destroy(cl);
  return 0;
}
```

An RPC Example

- You must first create a client structure using **clnt_create()**

```
#include <rpc/rpc.h>
CLIENT *clnt_create(char *host, u_long prog, u_long vers, char *proto);
```

  - **host**: the name of the server machine
  - **prog**, **vers**: the program and version numbers
  - **proto**: the transport protocol to use (“tcp”, or “udp”)

- Then you can call the (generated) client procedure **add_proc_1()** to send the RPC

- When you are finished, you destroy the client structure
  - A client structure can be used multiple times without being destroyed and re-created

An RPC Example

- Compile your client

```
$ gcc -c client.c
$ gcc -c add_clnt.c
$ gcc -c add_xdr.c
$ gcc -o client client.o add_clnt.o add_xdr.o -lnsl
```

- Try it all
  - Start your server

```
$ ./server
```

  - Send a request

```
$ ./client flits.cs.vu.nl 30 7
We received the result: 37
```

Concurrent RPC Servers

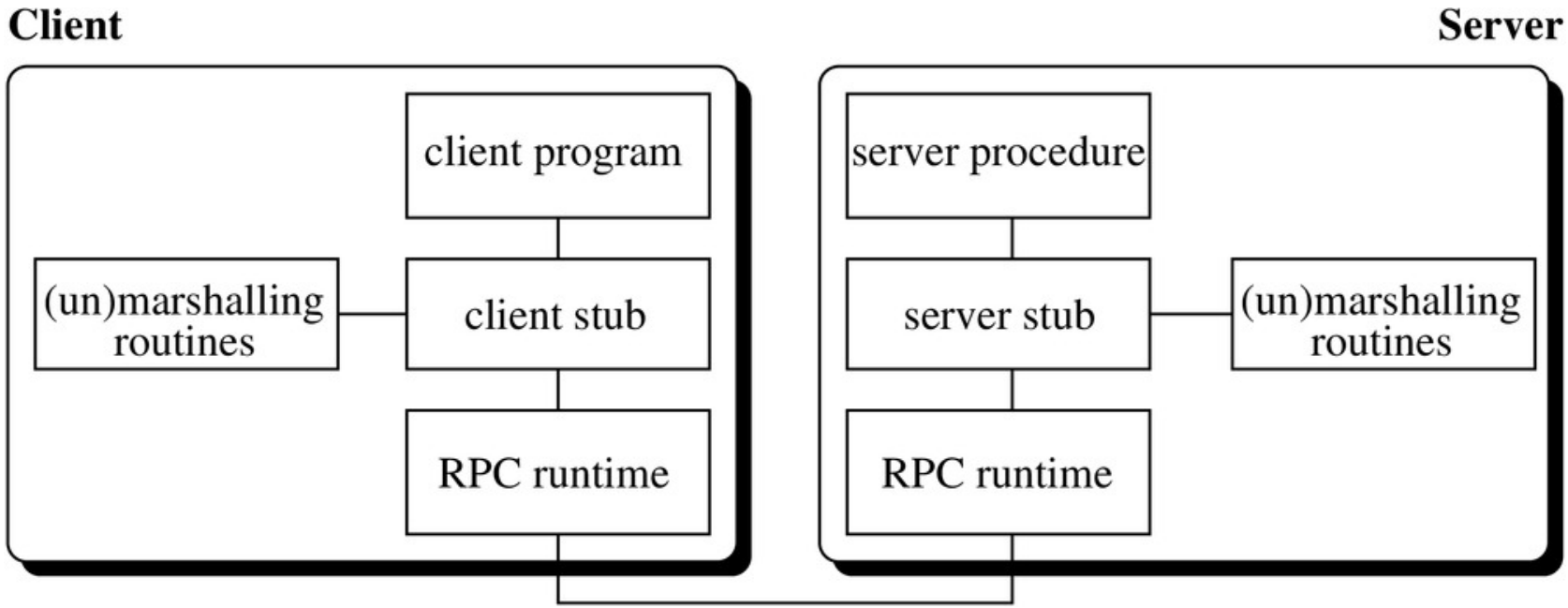
- By default, generated RPC servers are iterative
  - **QUESTION:** how can you check that by yourselves?

- Some versions of **rpcgen** allow one to generate server code which implements one-thread-per-client
  - Solaris has a multi-thread extension, Linux doesn't
  - On Solaris: use rpcgen –A –M instead of rpcgen to generate a multithreaded server
  - You also need to change your client program and server procedure

- **QUESTION:** Can you transform an iterative RPC server into a concurrent one by adding (for example) `fork()` calls in your server procedure?

Inside Sun RPC

The Global Picture



Low Level RPC

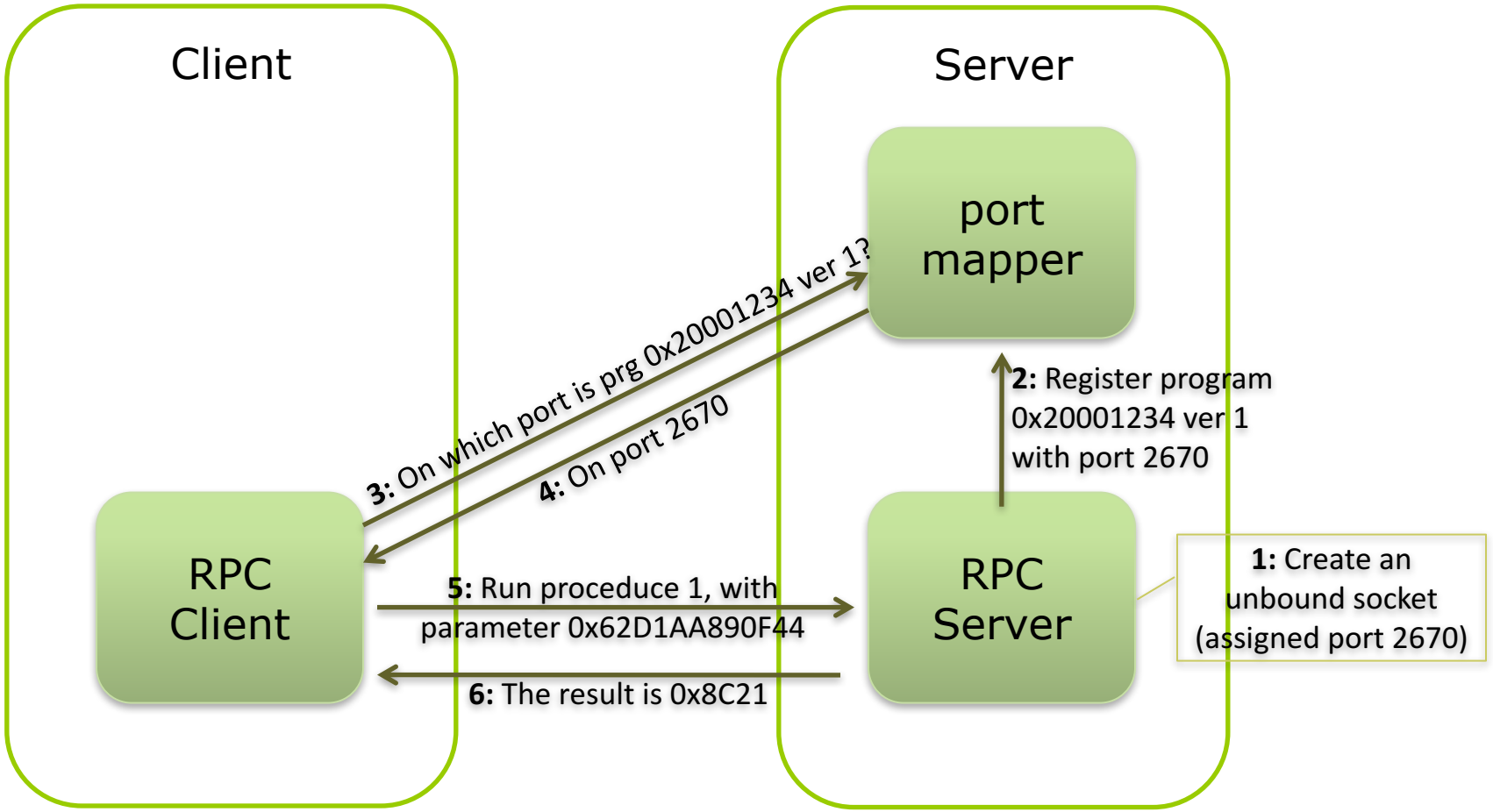
The Port Mapper [1/3]

- Did you notice that we did not specify a port number for our *add* server?
  - This could be done by using a well-known port number for all RPCs
  - But in that case, we would not be able to run multiple servers in the same machine simultaneously

- The RPC solution: **The Port Mapper**
  - The port mapper is a server running on **port 111** (well-known port)
  - When your RPC server starts, it does not bind its socket to a specific port. Instead, it registers whatever port has been assigned to it by the OS to the local port mapper
  - When you create a client with **clnt_create**, you automatically contact the remote port mapper to ask for the port number of the server you want to contact

Low Level RPC

The Port Mapper [2/3]



Low Level RPC

The Port Mapper [3/3]

- You can also query a port mapper by hand:

```
$ rpcinfo -p acropolis.cs.vu.nl
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
  55848960    1   udp  32898
  55848960    1   tcp  58411
$
```

Low Level RPC

A Glimpse of the Protocol

- The RPC runtime does not know anything about **procedure names** or **prototypes**
- All it does is manage **requests** containing:
  - A host name
  - A program number
  - A version number
  - A procedure number
  - A buffer containing the arguments
- And **responses** containing:
  - A buffer with the return values
- The RPC runtime does not know (nor does it care) what the procedures are doing or what kind of arguments they take
  - It just ships buffers

Low Level RPC

Transport Issues

- RPC invocation can be carried out either by **UDP** or **TCP**
- **UDP** is **lighter**, **faster**
  - But what happens if the **request gets lost**?
    - After a timeout, the client system automatically resends the request, until it receives a response
  - Now, what happens if the **response gets lost**?
    - The server has been invoked twice
    - **UDP-based RPC** implements the **At-Least-Once semantics**: each invocation is performed on the server at least once, but sometimes several times
- You must be aware of these issues
  - If you choose **UDP**, then your server must be **idempotent** (i.e., applying the same operation twice has the same effect as applying it once)
  - If your server is not idempotent, then you **must** use **TCP**
    - **QUESTION:** Why would UDP not work?
    - **QUESTION:** How could we make it work for non-idempotent services?

Medium Level RPC

(Un)Marshalling Parameters

- Parameters must be converted from variables into buffers and vice-versa
  - The lowest level only knows about parameters in the form of buffers to be shipped
  - The user programs only know about parameters in the form of C variables
  - The XDR (e**X**ternal **D**ata **R**epresentation) layer is in charge of the conversion
- The XDR buffer representations must be architecture independent
  - To avoid problems when a Little-Endian client sends requests to a Big-Endian server, or vice versa

Medium Level RPC

(Un)Marshalling Parameters

- The XDR library contains basic encoding/decoding functions
- Example: to (un)marshal an integer

```
#include <rpc/xdr.h>
bool_t xdr_int(XDR *xdrs, int *ip);
```

  - The xdrs structure contains a file descriptor of the socket where to read/write the buffers
  - It also contains a flag to indicate whether xdr_int must encode or decode the buffer
    - Encode: convert *ip into a message buffer and write it to the socket
    - Decode: read a message buffer from the socket and convert it to *ip

Medium Level RPC

(Un)Marshalling Parameters

- rpcgen generates functions to (un)marshal the input and output parameters of your RPC procedures:

```
#include "add.h"

bool_t xdr_add_in(XDR *xdrs, add_in *objp)     /* to convert add_in */
{
  if (!xdr_long(xdrs, &objp->arg1)) return FALSE;
  if (!xdr_long(xdrs, &objp->arg2)) return FALSE;
  return TRUE;
}

bool_t xdr_add_out(XDR *xdrs, add_out *objp)   /* to convert add_out */
{
  if (!xdr_long(xdrs, objp)) return FALSE;
  return TRUE;
}
```

High Level RPC

Client Stub

- The client stub generated by rpcgen is in charge of providing a simple interface to the client programs:
  - It calls the XDR functions to marshal the procedure parameters and to unmarshal the return values
  - It ships them to the low-level RPC layers to be transferred to the server

```
add_out *add_proc_1(add_in *argp, CLIENT *clnt) {
  static add_out clnt_res;

  memset((char *)&clnt_res, 0, sizeof(clnt_res));
  if (clnt_call(clnt, ADD_PROC,
                (xdrproc_t) xdr_add_in, (caddr_t) argp,
                (xdrproc_t) xdr_add_out, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
    return (NULL);
  }
  return (&clnt_res);
}
```

High Level RPC

Server Skeleton

- The server skeleton generated by **rpcgen** does the following:
  1. Defines a function to reply to any request directed at it (same program and version number, any procedure number)
     - It checks the procedure number of the incoming request
     - Calls the appropriate function to unmarshal the incoming parameters
     - Calls the user procedure
     - Marshals the return values
  2. Creates a **UDP** **and** a **TCP** server socket, and registers them to the port mapper
  3. Registers the function defined in step 1 to the RPC system:
     - Program and version numbers
     - The UDP and TCP ports
     - The procedure to serve incoming requests
  4. Waits for RPC requests to arrive

CORBA

(not included in the exam material, read optionally)

CORBA

- CORBA
  - **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
  - A remote object invocation system

- Language-independent, Architecture independent
  - e.g., a C++ object on a Linux computer can invoke a Java object on a Windows computer

- Corba is a specification (not a *piece of software*)
  - Corba defines the protocols, the interfaces, etc.
  - Any vendor can implement the specification
    - An implementation is called an ORB (**O**bject **R**equest **B**roker)
  - Each ORB can contain additional tools to help developers
    - Some are standardized, some not
  - There are many ORBs
    - For instance, the JDK-1.2 contains an ORB

The Interface Definition Language

- Since CORBA is language-independent, you must represent server interfaces in a way that every language can understand: the **I**nterface **D**efinition **L**anguage (IDL)

- Example:

```
module Stock {
  exception Invalid_Stock {};

  struct Info {
    long high;
    long low;
    long last;
  };

  interface Quoter {
    Info get_quote(in string stock_names) raises(Invalid_Stock);
  };
};
```

IDL Compilers

- Each ORB provides an IDL compiler
  - It compiles the interface specification into stubs and skeletons
  - Example: the JDK-1.2 ORB contains an IDL compiler called **idltojava**, or **idlj**

- The CORBA specification defines language mappings
  - How to represent an IDL interface in a given language
  - Two ORBs must generate exactly the same language specific interface from the same IDL
  - There are mappings for C, C++, Java, Python, Perl, Ruby, Ada, COBOL, Smalltalk, Objective-C, Lisp, etc.

Example: The Java Mapping

- The IDL concepts are translated into Java concepts:

IDL	Java
Module	Package
Interface	Interface
Operation	Method
struct	Object
etc.	

Implementing a Server and a Client

- Once you have compiled your IDL interface, you can fill in the implementation for your server object
  - Details vary by language (of course)

- To write a server program
  1. Initialize the ORB
  2. Instantiate (at least) one server object
  3. Connect the server object to the ORB
  4. Wait for requests

- To write a client program
  1. Initialize the ORB
  2. Obtain a reference to a server object
  3. Invoke the method

Cross-language Remote Invocations

- To send requests from a client written in language A to a server written in language B:
  1. Write an IDL specification of the server
  2. Compile the specification to language B, implement the server in language B
  3. Compile the specification to language A, implement the client in language A
  4. Run it!

Object Adapters

- An object adapter is in charge of managing server objects
  - Can server objects be invoked concurrently? Each object can decide which policy it prefers
  - Sometimes you want to store server objects on disk, and instantiate them only when a request is directed to it
  - etc.

- The Portable Object Adapter (POA) is a standardized interface to such an Object Adapter

The Dynamic Invocation Interface

- Sometimes you don't know the exact interface of the server at compile time
- The Dynamic Invocation Interface (DII) allows you to dynamically construct a request before invoking it
- Beware: this code probably doesn't work

```
org.omg.CORBA.Object obj = ...;
org.omg.CORBA.Request req = obj->getRequest("set_quote");
req.addInArg("VU");
req.addInArg(42);
req.setReturnType(new org.omg.CORBA.int());
req->invoke();
int result = req.return_value();
```

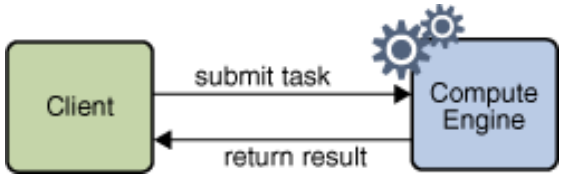
Corba Services

- In addition to basic remote invocation, CORBA defines services:
  - A CORBA service is a complete server object which is provided by an ORB
  - They provide high-level functions to help you develop applications
- Examples
  - The Naming Service associates a name with an object
  - The Trading Service allows you to search for an object by functionality
  - The Event Service allows you to be notified when certain events happen
  - The Interface Repository allows you to ask “What is the interface of this object?”

RMI
Remote Method Invocation

Java RMI

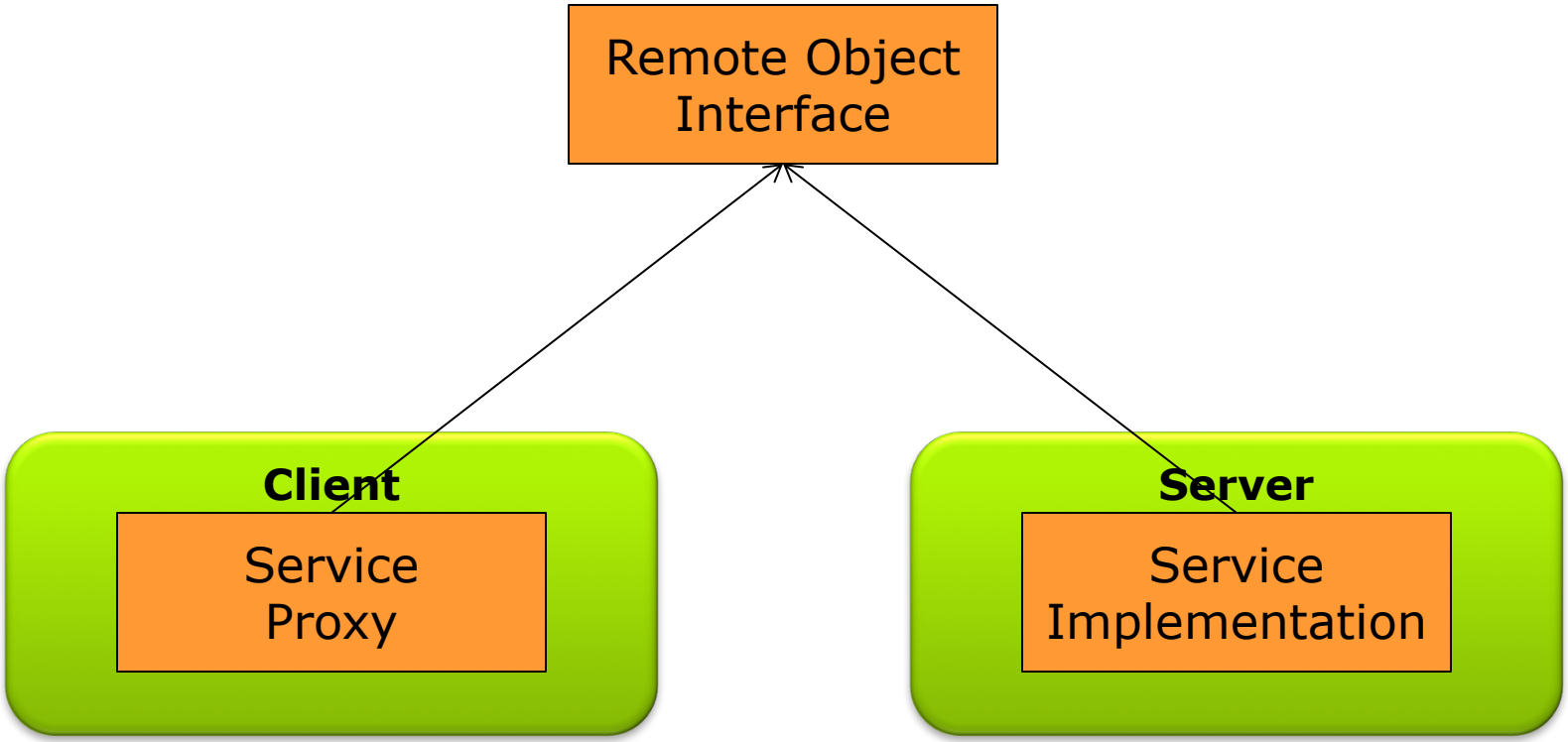
- RMI (Remote Method Invocation)
  - allows Java programs to invoke methods of remote objects



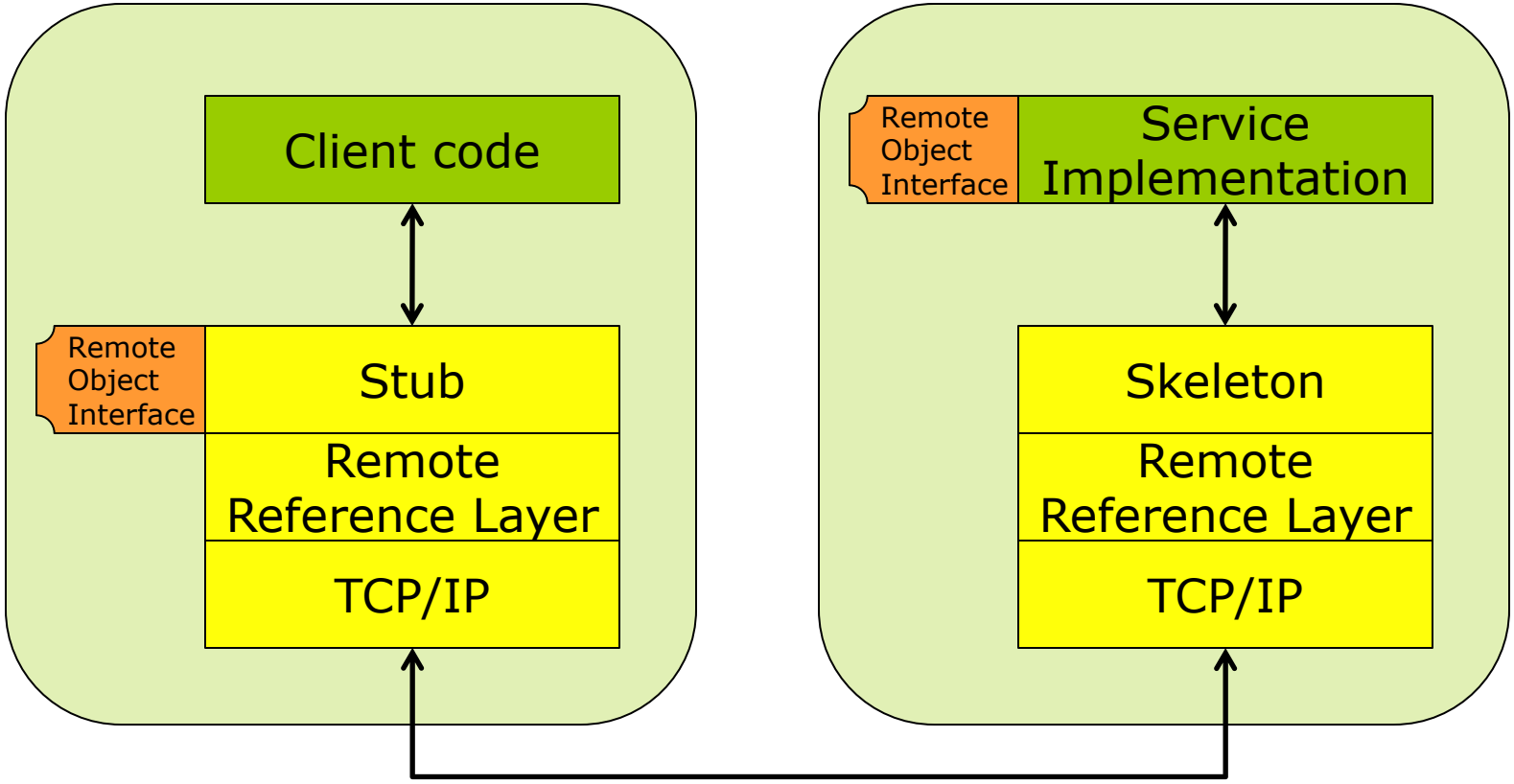
- Access distributed objects (almost) identically to local objects
  - Same syntax (arguments, return values, etc.)
  - Same semantics (exceptions)

- Scope
  - Only in Java
  - Significant changes in Java 1.2 (incompatible to earlier RMI)

RMI Interface



RMI Architecture



Transparency

- To the client, a remote object appears exactly like a local object
  - Except that **you must bind it first**!
- This is possible thanks to **interfaces**
  - You write the **interface** to the remote object
  - You write the **implementation** for the remote object
  - RMI automatically creates a **stub class** which implements the remote object interface
  - The client accesses the stub exactly the same way it would access a local copy of the remote object

The RMI Registry

- Java RMI needs a naming service (like Sun RPC's port mapper)
  - Servers register contact address information
  - Clients can locate servers
- This is called **RMI Registry**
- Unlike in RPC, you must start the RMI Registry yourself
  - **Question:** Where?
    - on each machine that hosts **server objects**
  - It is a program called **`rmiregistry`**
  - By default runs on port 1099 (but you can specify another port number: **`rmiregistry <port_nb>`**)
- Programs access the registry via the **`java.rmi.Naming`** class

An RMI Example

- To write a minimalist RMI program you must write:
  - An interface for the remote object: **Remote.java**
  - An implementation for the remote object: **RemoteImpl.java**
  - A server which will run the remote object: **RemoteServer.java**
  - A client to access the server: **RemoteClient.java**

- The RMI compiler **`rmic`** will generate the rest:
  - For Java version up to 1.1
    - A client stub: **RemoteImpl_Stub.class** (already compiled)
    - A server skeleton: **RemoteImpl_Skel.class** (already compiled)
  - For Java version 1.2 … 1.4 (Java 2 … Java 4)
    - A single stub, used for both client and server: **RemoteImpl_Stub.class**
  - For Java 1.5 (Java 5) and higher
    - Nothing is needed!
    - Stubs and skeletons are included in the Java library

An RMI Example (continued)

- Start by writing the interface for the remote object: **Calculator.java**

```
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;
}
```

- You must respect a few rules
  - The interface must extend the **java.rmi.Remote** interface
  - All methods must throw the **java.rmi.RemoteException** exception

- Compile the interface

```
$ javac Calculator.java
```

An RMI Example (continued)

- Write an implementation for the remote object: CalculatorImpl.java

```
public class CalculatorImpl
  extends java.rmi.server.UnicastRemoteObject
  implements Calculator {

  // Implementations must have an explicit constructor
  public CalculatorImpl() throws java.rmi.RemoteException {
    super();
  }

  public long add(long a, long b) throws java.rmi.RemoteException {
    return a + b;
  }

  public long sub(long a, long b) throws java.rmi.RemoteException {
    return a - b;
  }
}
```

An RMI Example (continued)

- The implementation class must respect a few constraints
  - It must implement the interface (of course!)
  - It must inherit from the **java.rmi.server.UnicastRemoteObject** class
  - It must have an explicit constructor which throws the **java.rmi.RemoteException** exception

- Compile the implementation class

```
$ javac CalculatorImpl.java
```

An RMI Example (continued)

- If you are using Java version earlier than 5.0, generate the stub and skeleton:

```
$ rmic CalculatorImpl
```

- This generates directly the **CalculatorImpl_Stub.class** and (for Java < 2.0) the **CalculatorImpl_Skel.class** files
  - Already compiled to byte code, you are not expected to run javac

An RMI Example (continued)

- Write a server program: CalculatorServer.java

```
import java.rmi.Naming;

public class CalculatorServer {
   public CalculatorServer() {
     try {
       Calculator c = new CalculatorImpl();
       Naming.rebind("rmi://localhost/CalculatorService", c);
     } catch (Exception e) {
       System.out.println("Trouble: " + e);
     }
   }

   public static void main(String args[]) {
     new CalculatorServer();
   }
}
```

An RMI Example (continued)

- The server program creates a **CalculatorImpl** object
- It registers the object to the local RMI registry

```
rebind(String name, Remote obj)
```

  - Associates a **name** to an **object**
  - Names are in URL form: **rmi://<host_name>[:port]/<service_name>**

- The server will wait for incoming requests
- Compile your server

```
$ javac CalculatorServer.java
```

An RMI Example (continued)

- Write a client program: CalculatorClient.java

```
import java.net.MalformedURLException;
import java.rmi.Naming;

public class CalculatorClient {
  public static void main(String[] args) {
    try {
      Calculator c = (Calculator)Naming.lookup(
                          "rmi://flits.few.vu.nl/CalculatorService");
      System.out.println( c.add(4, 5) );
      System.out.println( c.sub(4, 3) );
    }
    catch (Exception e) {
      System.out.println("Received Exception:");
      System.out.println(e);
    }
  }
}
```

An RMI Example (continued)

- Before invoking the server, the client must lookup the registry
  - It must provide the URL for the remote service
  - It gets back a stub which has exactly the same interface as the server
  - It can use it as a local object: **long x = c.add(4,5)**

- Compile your client:

```
$ javac CalculatorClient.java
```

An RMI Example (end!)

- Try your program!!
  - Start the RMI Registry: rmiregistry

```
$ rmiregistry &
```

    - The registry must have access to your classes
    - Either start the registry in the same directory as the classes, or make sure the directory is listed in the $CLASSPATH variable

  - Start your server

```
$ java CalculatorServer
```

  - Start your client

```
$ java CalculatorClient
9
1
$
```

Using RMI in a Distributed Context

- First, try your RMI program on a single host
  - It will work only if there is an operational TCP configuration

- To use the client and server on different hosts, you must provide the right files to the right entities:
  - The **server** and the **rmiregistry** need the following files:
    - **Calculator.class**: the remote object interface
    - **CalculatorImpl.class**: the server object implementation
    - **CalculatorServer.class**: the server program
  - The **client** needs the following files:
    - **Calculator.class**: the remote object interface
    - **CalculatorClient.class**: the client program

Passing Parameters

- Depending on the parameter's type, a different method is followed
- **Primitive types** (e.g., int, float, long, char, boolean) are copied by value
- **Object types** are serialized and transferred
  - The object itself, plus every other object that it refers to (recursively)
  - You must be careful:
    - Remote invocations **pass objects by value** (whereas local invocations pass objects **by reference**)
    - It is very easy to transfer huge quantities of data without noticing
      - (e.g., you have a whole database in memory, and you transfer an object that contains a reference to the database)
- **Remote Objects** (i.e., inheriting from java.rmi.Object) are not transferred!
  - Instead, a distributed reference to this object is transferred
  - Any invocation to this object will result in an RMI request

Questions

- Can you also operate RMI over UDP?
  - No! Only TCP.
- Why does the main thread in CalculatorServer.java end?
  - A Java program terminates when all threads have terminated, unlike a C program that terminates when the main thread terminates
- bind() vs. rebind() ?
- When do we get exceptions?
  - Exceptions raised on the server are propagated and thrown on the client

Questions

- Is RMI blocking on the client side?
  - YES: The client blocks until the server replies
- How can we make RMI non-blocking for the client's side?
  - **Option 1**: Make separate thread and let it block
  - **Option 2**: Or make the client also a Remote Object, pass it on to the server, and let it call you back once it's done
- Is the server iterative?
  - No!
  - Two or more requests can be handled simultaneously

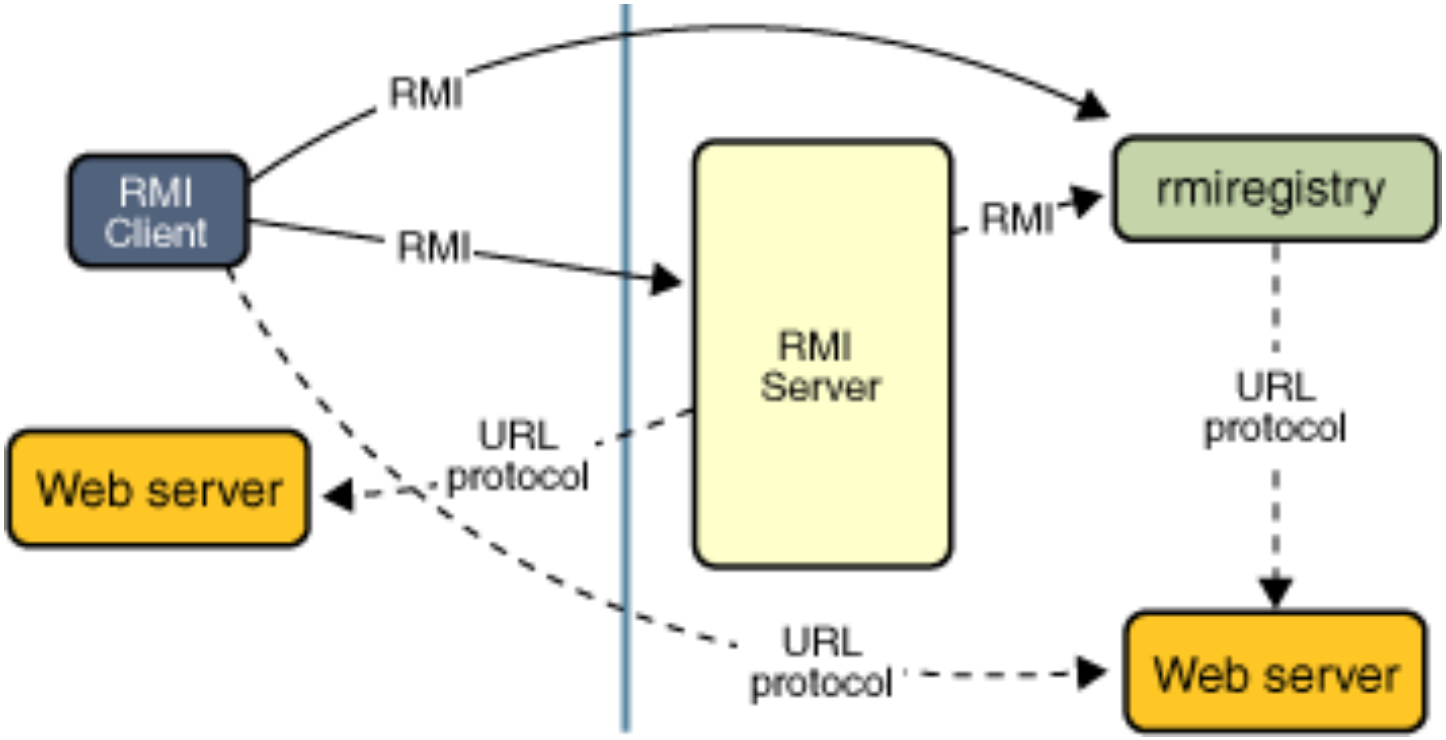
Remote Codebase [1/2]

- Why does the Registry require access to the implementation files?
  - Because a JVM can send to another JVM a (serializable) object whose class is not known by the receiver (although its interface *is* known)

- What if we are using a remote registry? (i.e., running on a different machine than the one hosting the remote object)
  - You can specify a remote codebase for Foo like this:

```
$ java -Djava.rmi.server.codebase=http://myhost/~abc/myclasses/ Foo
```

Remote Codebase [2/2]



Security

- When a Java application needs to download code of a serializable class from a remote location, security has to be considered
- To set your custom security policy, start your application like this:

```
$ java -Djava.security.policy=myPolicy.txt Foo
```

- If your program crashes with a security exception, try it out (temporarily) with a policy file like this:

```
grant {
  // Allow everything for now
  permission java.security.AllPermission;
};
```

- The default policy file is found at **<JAVAHOME>/lib/security/java.policy**

Garbage Collection

- Garbage Collection is hard in a single machine…
  - …very hard in a distributed setting!
- The server keeps a reference counter on
  - local references for a Remote Object
  - client references for a Remote Object
- The server garbage collects Remote Objects that are not referenced locally nor remotely
- A Remote Object can implement the **java.rmi.server.Unreferenced** interface to get a notification via the **unreferenced()** method when all clients have dropped it

Application Architecture Example

- Discussion in class:
  - How would you implement a Multiplayer Chess system by RMI?

Remote Codebase [2/2]