LAU
الجامعة اللبنانية الأميركية
Lebanese American University

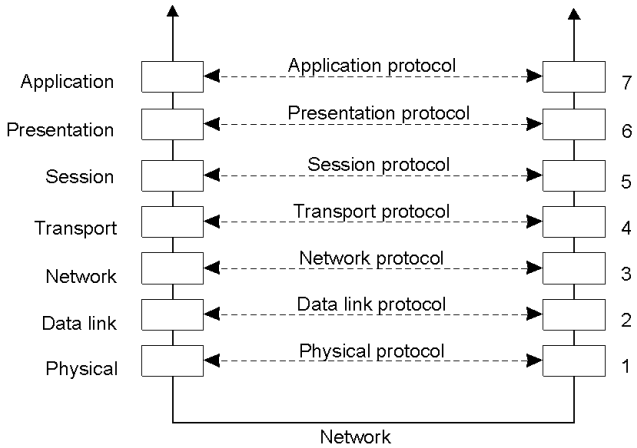
CSC 634: Networks Programming

Lecture 07: More Client-Server Architectures, RPC, RMI

Instructor: Haidar M. Harmanani

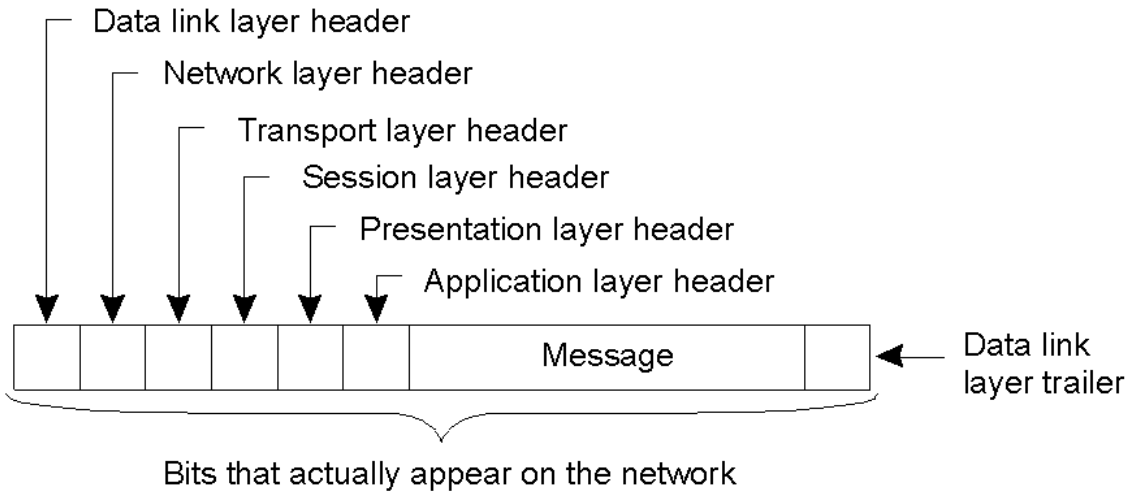
Communication Protocols

- Protocols are agreements/rules on communication
- Protocols could be connection-oriented or connectionless

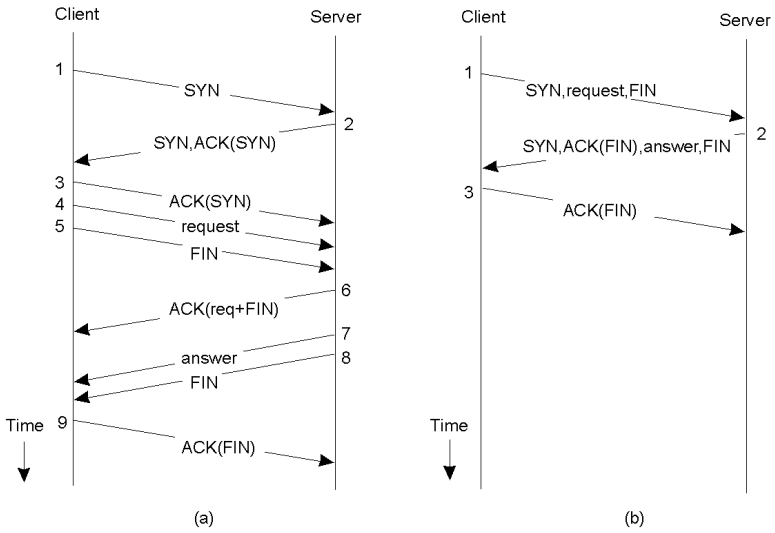


Layered Protocols

- A typical message as it appears on the network.



Client-Server TCP



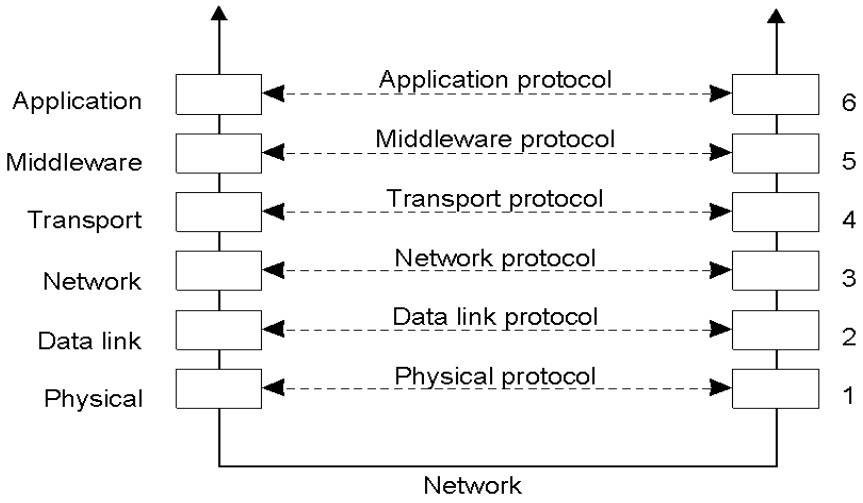
Client-Server Programming

- Various issues need to be dealt with:
  - How to structure the program
  - Define an application protocol that is powerful enough for the specific needs yet efficiently implemented
  - Deal with machines of different architectures
    - They may represent data differently
  - Locate machines
    - "Which machine implements task X?"
  - etc.

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

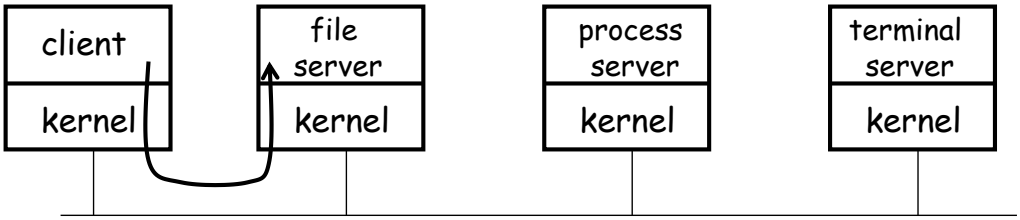
Middleware Protocols

- Middleware: layer that resides between an OS and an application
  - May implement general-purpose protocols that warrant their own layers
    - Example: distributed commit



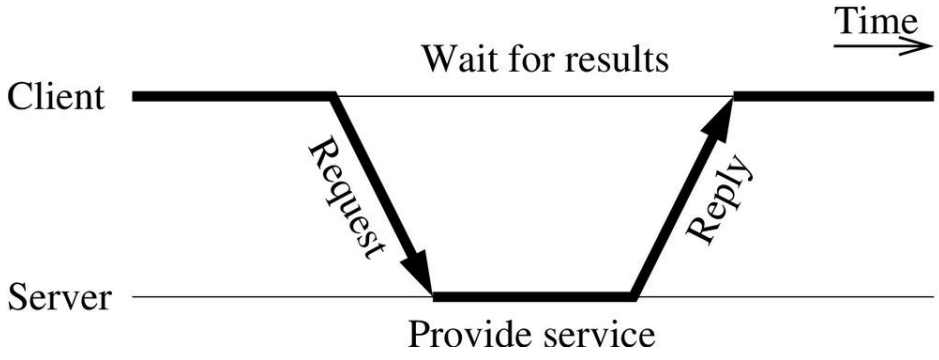
Client-Server Communication Model

- Structure: group of servers offering service to clients
- Based on a request/response paradigm
- Techniques:
  – Sockets, remote procedure calls (RPC), Remote Method Invocation (RMI)



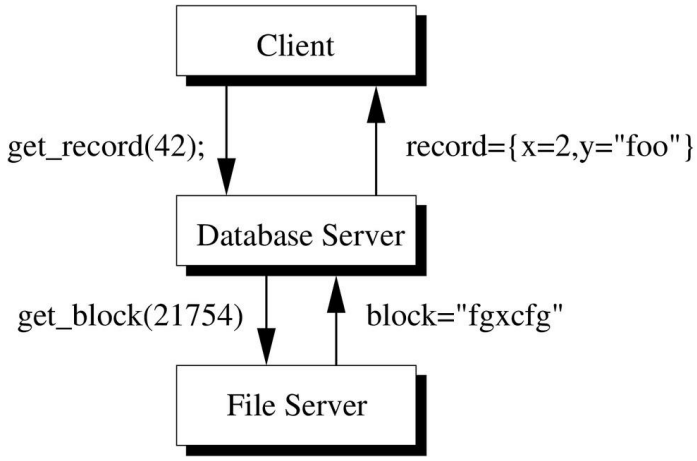
The Client-Server Model

- 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:
  - Just have to be careful regarding loops



Why use the Client-Server Model?

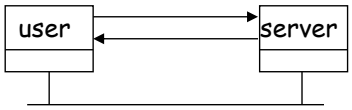
- A client-server application (usually) does not run faster than a centralized application
  - The client is waiting while the server works
  - Additional delays due to communication
- But it has several advantages!
  - Benefitting from **specialized resources**
    - Fast CPU, lots of memory, etc.
    - 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.

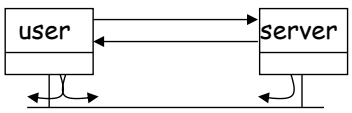
Issues in Client-Server Communication

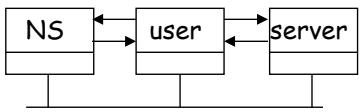
- Addressing
- Blocking versus non-blocking
- Buffered versus unbuffered
- Reliable versus unreliable
- Server architecture: concurrent versus sequential
- Scalability

Addressing Issues

- Question: how is the server located?
- Hardwired address
  - Machine address and process address are known a priori
- Broadcast-based
  - Server chooses address from a sparse address space
  - Client broadcasts request
  - Can cache response for future
- Locate address via name server





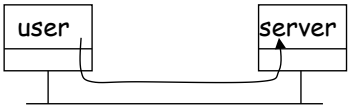


Blocking versus Non-blocking

- Blocking communication (synchronous)
  - Send blocks until message is actually sent
  - Receive blocks until message is actually received
- Non-blocking communication (asynchronous)
  - Send returns immediately
  - Return does not block either

Buffering Issues

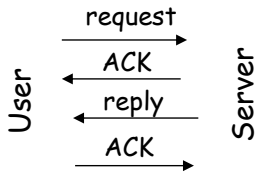
- Unbuffered communication
  - Server must call receive before client can call send



- Buffered communication
  - Client send to a mailbox
  - Server receives from a mailbox

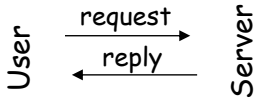
Reliability

- Unreliable channel
  - Need acknowledgements (ACKs)
  - Applications handle ACKs
  - ACKs for both request and reply



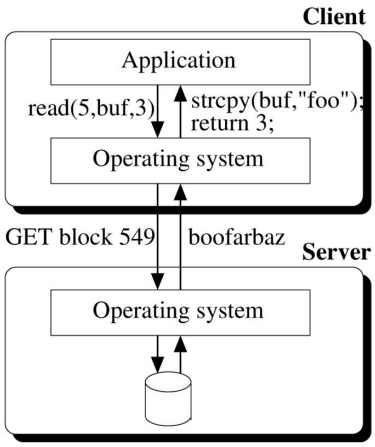
- Reliable channel
  - Reply acts as ACK for request

- Reliable communication on unreliable channels
  - Transport protocol handles lost messages



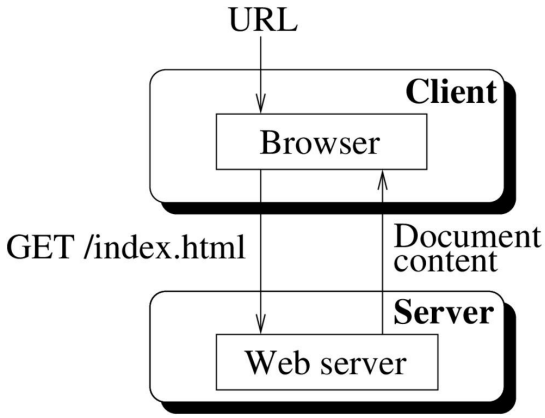
Example

- **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



Server Design Issues

- Server Design
  - Iterative versus concurrent
- How to locate an end-point (port #)?
  - Well known port #
  - Directory service (port mapper in Unix) – Super server (inetd in Unix)
- Stateful server
  - Maintain state of connected clients – Sessions in web servers
- Stateless server
  - No state for clients
- Soft state
  - Maintain state for a limited time; discarding state does not impact correctness

Communication Between Processes

- Unstructured communication
  – Use shared memory or shared data structures
- Structured communication
  – Use explicit messages
- Distributed Systems: both need low-level communication support

Server Clusters

- Web applications use tiered architecture
  – Each tier may be optionally replicated; uses a dispatcher
  – Use TCP splicing or handoffs

Server Architecture

- Sequential
  - Serve one request at a time
  - Can service multiple requests by employing events and asynchronous communication
- Concurrent
  - Server spawns a process or thread to service each request
  - Can also use a pre-spawned pool of threads/processes (apache)
- Thus servers could be
  - Pure-sequential, event-based, thread-based, process-based
- Discussion: which architecture is most efficient?

Scalability

- *Question:* How can you scale the server capacity?
  - Buy bigger machine!
  - Replicate
  - Distribute data and/or algorithms
  - Ship code instead of data
  - Cache
  - Get a scalable cloud solution

To Push or Pull ?

- Client-pull architecture
  - Clients pull data from servers (by sending requests)
    - Example: HTTP
  - Pro: stateless servers, failures are each to handle
  - Con: limited scalability
- Server-push architecture
  - Servers push data to client
  - Example: video streaming, stock tickers
  - Pro: more scalable
  - Con: stateful servers, less resilient to failure
- When/how-often to push or pull?

Group Communication

- One-to-many communication: useful for distributed applications
- Issues:
  - Group characteristics:
    - Static/dynamic, open/closed
  - Group addressing
    - Multicast, broadcast, application-level multicast (unicast)
  - Atomicity
  - Message ordering
  - Scalability

Putting it all together: Email

- User uses mail client to compose a message
- Mail client connects to mail server
- Mail server looks up address to destination mail server
- Mail server sets up a connection and passes the mail to destination mail server
- Destination stores mail in input buffer (user mailbox)
- Recipient checks mail at a later time
- Recently, server-push architecture have become very common in email systems

Email: Design Considerations

- Structured or unstructured?
- Addressing?
- Blocking/non-blocking?
- Buffered or unbuffered?
- Reliable or unreliable?
- Server architecture
- Scalability
- Push or pull?
- Group communication

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

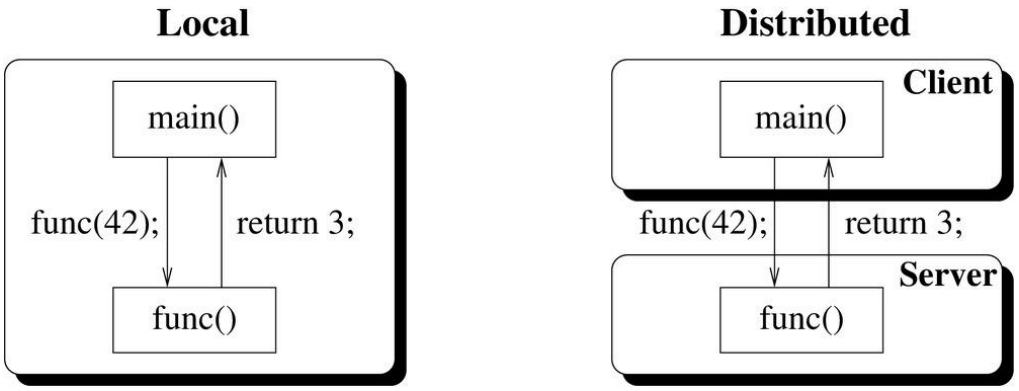
Remote Procedure Calls (RPC)

Remote Procedure Calls

- Goal: Make distributed computing look like centralized computing
- Allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language
- Issues
  - How to pass parameters
  - Bindings
  - Semantics in face of errors
- Two classes: integrated into prog language and separate

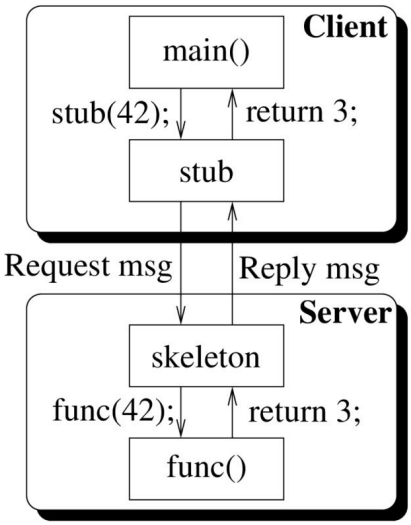
Remote Procedure Calls [1/2]

- 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

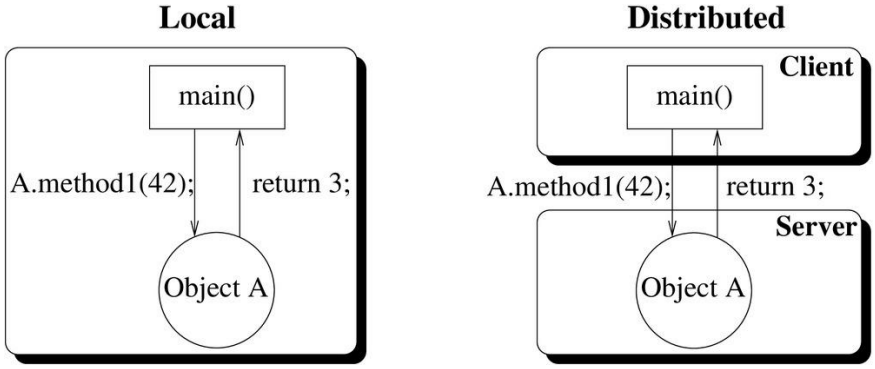
- Clients and servers do not share the same address space

- 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

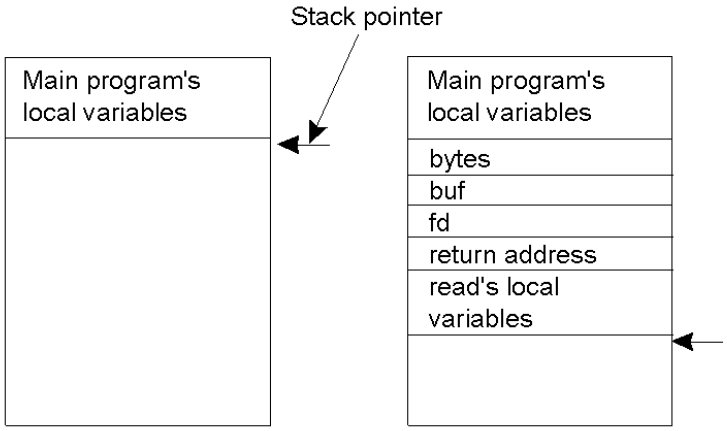
- The equivalent to RPC in the object-oriented world is RMI (Remote Method Invocation)
- Like in RPC, you must have stubs and skeletons
- There are several *Remote Method Invocation* systems:
  - Sun RMI (entirely in Java), Corba (language independent), etc.



Stubs: obtaining transparency

- Compiler generates from API stubs for a procedure on the client and server
- Client stub
  - Marshals arguments into machine-independent format
  - Sends request to server
  - Waits for response
  - Unmarshals result and returns to caller
- Server stub
  - Unmarshals arguments and builds stack frame
  - Calls procedure
  - Server stub marshals results and sends reply

Conventional Procedure Call



(a) (b)

Parameter passing in a local procedure call: the stack before the call to read

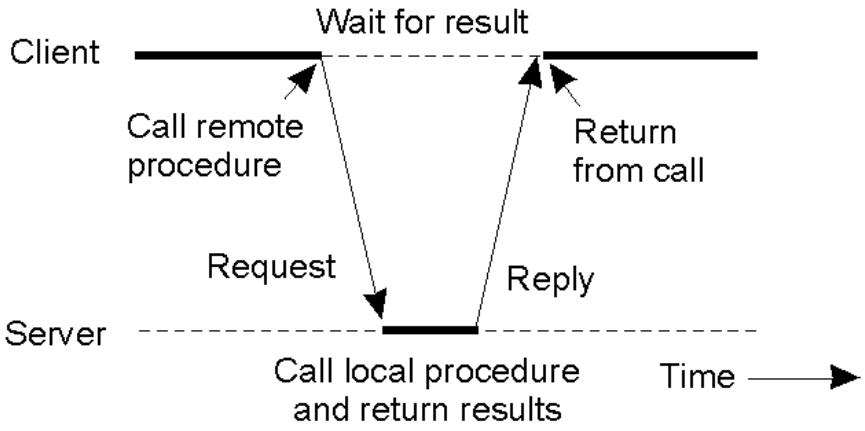
The stack while the called procedure is active

Parameter Passing

- Local procedure parameter passing
  - Call-by-value
  - Call-by-reference: arrays, complex data structures
- Remote procedure calls simulate this through:
  - Stubs – proxies
  - Flattening – marshalling
- Related issue: global variables are not allowed in RPCs

Client and Server Stubs

- Principle of RPC between a client and server program [Birrell&Nelson 1984]



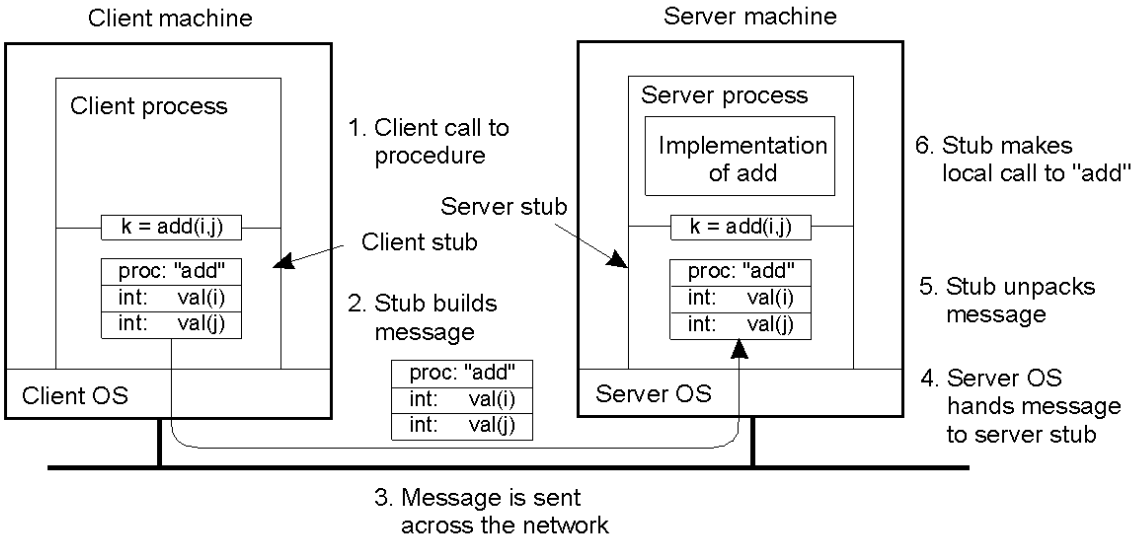
Stubs

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
  – Packaging parameters is called *marshalling*
- Stub compiler generates stub automatically from specs in an *Interface Definition Language (IDL)*
  – Simplifies programmer task

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Example of an RPC



Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a standard representation
  - Example: external data representation (XDR)
- Problem: how do we pass pointers?
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- What about data structures containing pointers?
  - Prohibit
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream

Binding

- Problem: how does a client locate a server?
  - Use Bindings
- Server
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder
- Client
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
    - Return handle and unique identifier to client

Binding: Comments

- Exporting and importing incurs overheads
- Binder can be a bottleneck
  - Use multiple binders
- Binder can do load balancing

Failure Semantics

- *Client unable to locate server*: return error
- *Lost request messages*: simple timeout mechanisms
- *Lost replies*: timeout mechanisms
  - Make operation idempotent
  - Use sequence numbers, mark retransmissions
- *Server failures:* did failure occur before or after operation?
  - At least once semantics (SUNRPC)
  - At most once
  - No guarantee
  - Exactly once: desirable but difficult to achieve

Failure Semantics

- *Client failure:* what happens to the server computation?
  - Referred to as an *orphan*
  - *Extermination*: log at client stub and explicitly kill orphans
    - Overhead of maintaining disk logs
  - *Reincarnation*: Divide time into epochs between failures and delete computations from old epochs
  - *Gentle reincarnation*: upon a new epoch broadcast, try to locate owner first (delete only if no owner)
  - *Expiration*: give each RPC a fixed quantum *T*; explicitly request extensions
    - Periodic checks with client during long computations

Implementation Issues

- Choice of protocol [affects communication costs]
  - Use existing protocol (UDP) or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
    - From client to NIC and from server NIC to server
  - As many as 7 copies
    - Stack in stub – message buffer in stub – kernel – NIC – medium – NIC – kernel – stub – server
  - Scatter-gather operations can reduce overheads

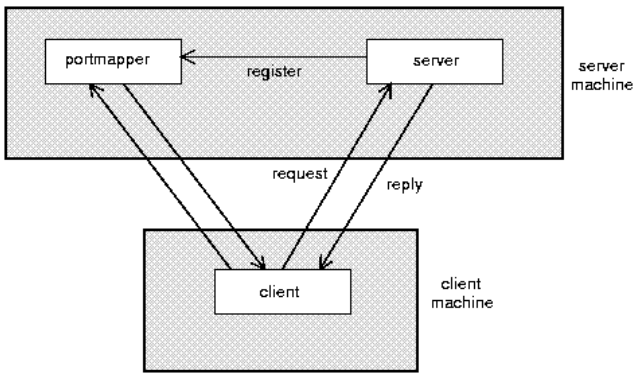
Case Study: SUNRPC

SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures

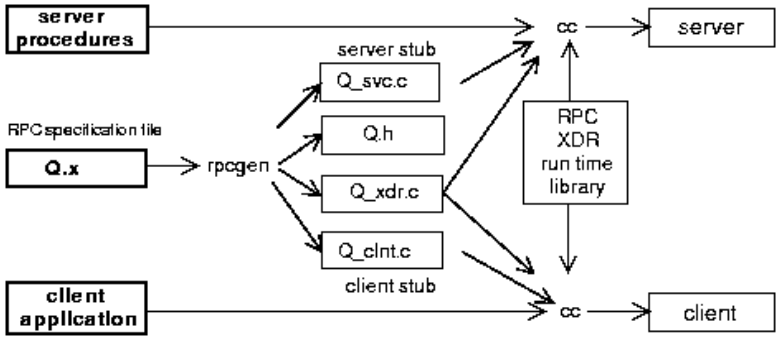
Binder: Port Mapper

- Server start-up: create port
- Server stub calls `svc_register` to register prog. #, version # with local port mapper
- Port mapper stores prog #, version #, and port
- Client start-up: call `clnt_create` to locate server port
- Upon return, client can call procedures at the server



Rpcgen: generating stubs

- Q_xdr.c: do XDR conversion
- Detailed example: later in this lecture



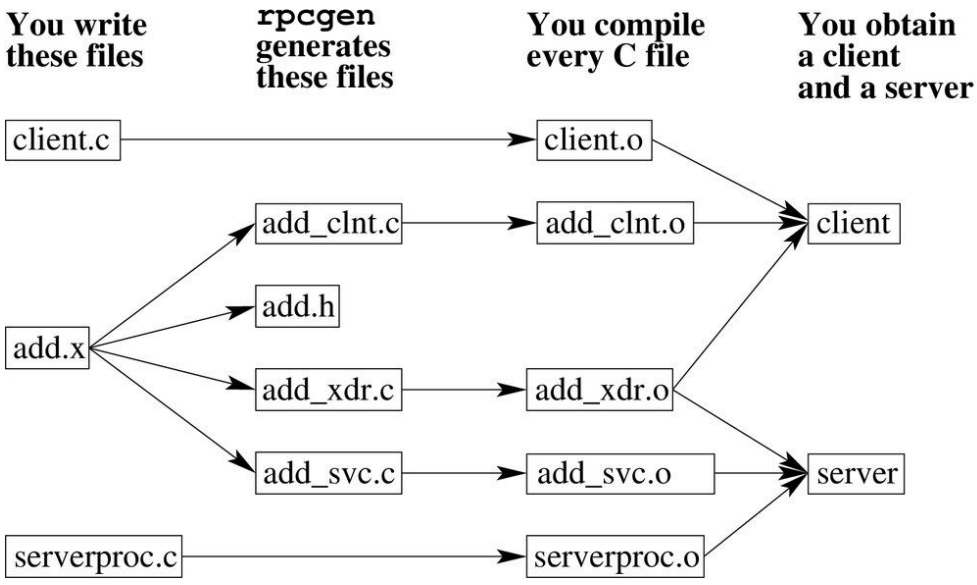
Writing an RPC Program [1/3]

- 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`

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
```

- Start the server

```
./server
```

An RPC Example: 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?

Lightweight RPCs

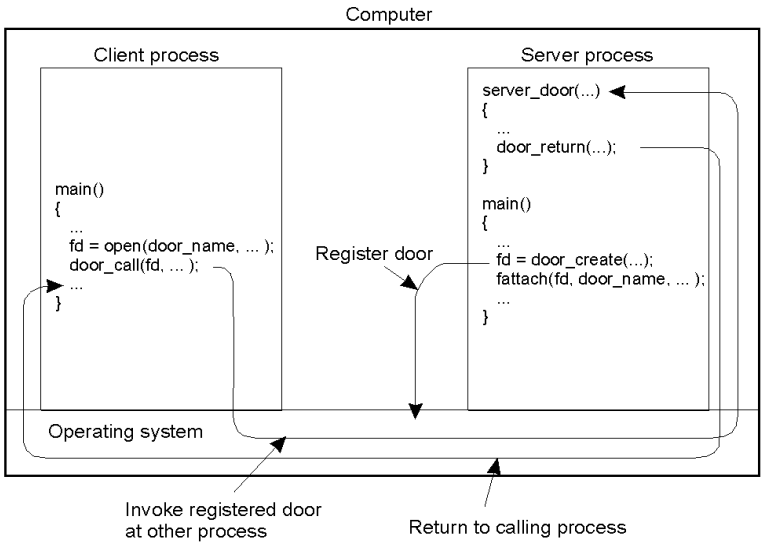
Lightweight RPCs

- Many RPCs occur between client and server on same machine
  – Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server *S* exports interface to remote procedures
- Client *C* on same machine imports interface
- OS kernel creates data structures including an argument stack shared between *S* and *C*

Lightweight RPCs

- RPC execution
  - Push arguments onto stack
  - Trap to kernel
  - Kernel changes mem map of client to server address space
  - Client thread executes procedure (OS upcall)
  - Thread traps to kernel upon completion
  - Kernel changes the address space back and returns control to client
- Called “doors” in Solaris

Doors



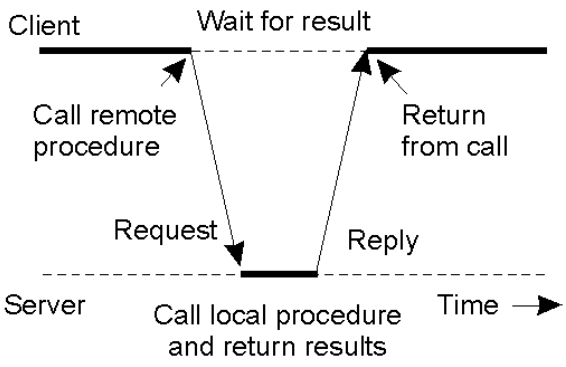
- Which RPC to use? - run-time bit allows stub to choose between LRPC and RPC

Other RPC Models

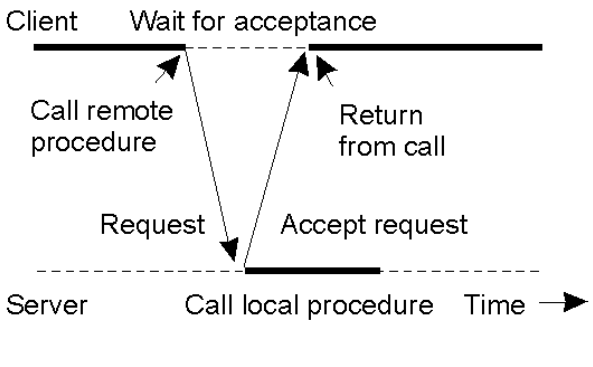
- Asynchronous RPC
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later
- Deferred-synchronous RPC
  - Use two asynchronous RPCs
  - Client needs a reply but can't wait for it; server sends reply via another asynchronous RPC
- One-way RPC
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).

Asynchronous RPC

- (a) The interconnection between client and server in a traditional RPC
- (b) The interaction using asynchronous RPC



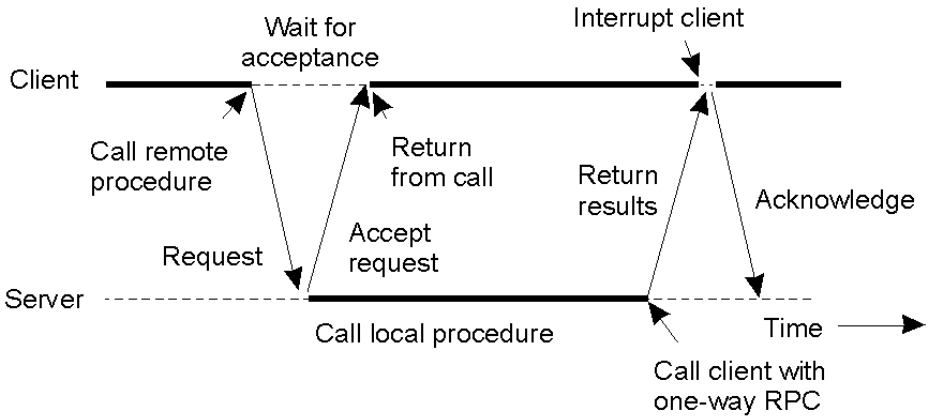
(a)



(b)

Deferred Synchronous RPC

- A client and server interacting through two asynchronous RPCs



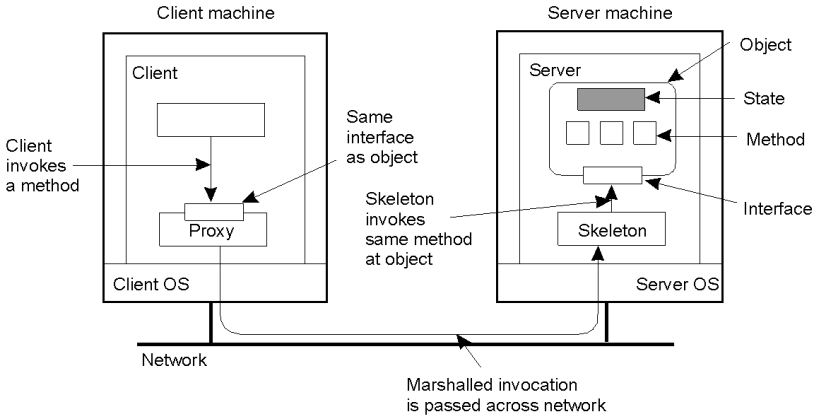
Remote Method Invocation

Remote Method Invocation (RMI)

- RPCs applied to *objects,* i.e., instances of a class
  - *Class:* object-oriented abstraction; module with data and operations
  - Separation between interface and implementation
  - Interface resides on one machine, implementation on another
- RMIs support system-wide object references
  - Parameters can be object references

Distributed Objects

- When a client binds to a distributed object, load the interface ("proxy") into client address space
  - Proxy analogous to stubs
- Server stub is referred to as a skeleton



Proxies and Skeletons

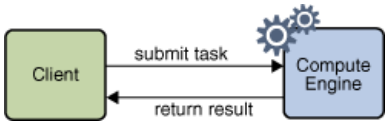
- Proxy: client stub
  - Maintains server ID, endpoint, object ID
  - Sets up and tears down connection with the server
  - [Java:] does serialization of local object parameters
  - In practice, can be downloaded/constructed on the fly (why can't this be done for RPCs in general?)
- Skeleton: server stub
  - Does deserialization and passes parameters to server and sends result to proxy

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Java RMI

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)

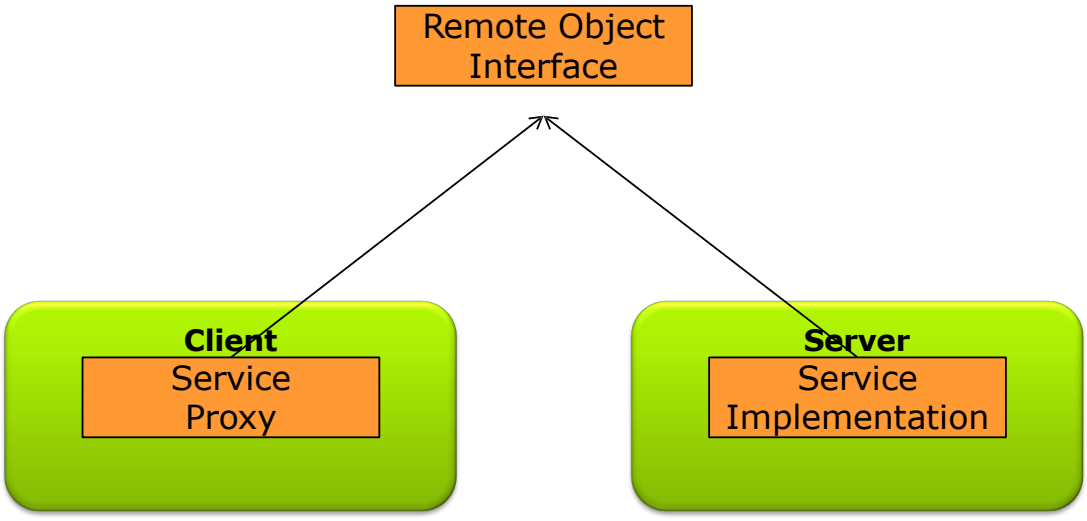
Java RMI

- Server
  - Defines interface and implements interface methods
  - Server program
    - Creates server object and registers object with “remote object” registry
- Client
  - Looks up server in remote object registry
  - Uses normal method call syntax for remote methods
- Java tools
  - Rmiregistry: server-side name server
  - Rmic: uses server interface to create client and server stubs

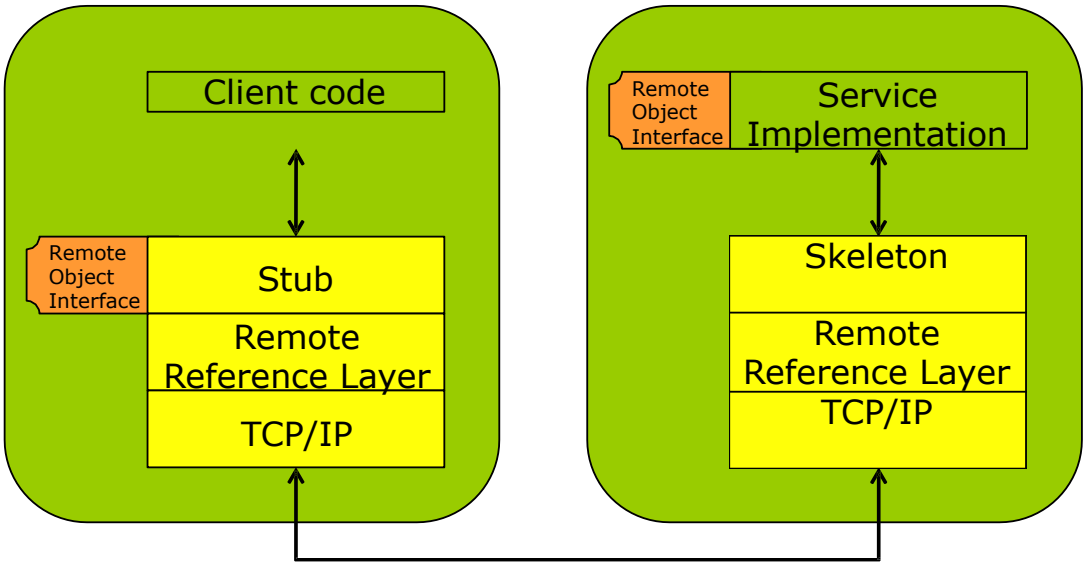
Java RMI and Synchronization

- Java supports Monitors: synchronized objects
  - Serializes accesses to objects
  - How does this work for remote objects?
- Options: block at the client or the server
- Block at server
  - Can synchronize across multiple proxies
  - Problem: what if the client crashes while blocked?
- Block at proxy
  - Need to synchronize clients at different machines
  - Explicit distributed locking necessary
- Java uses proxies for blocking
  - No protection for simultaneous access from different clients
  - Applications need to implement distributed locking

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

```
$ java CalculatorServer
```

- Start your server

```
$ 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

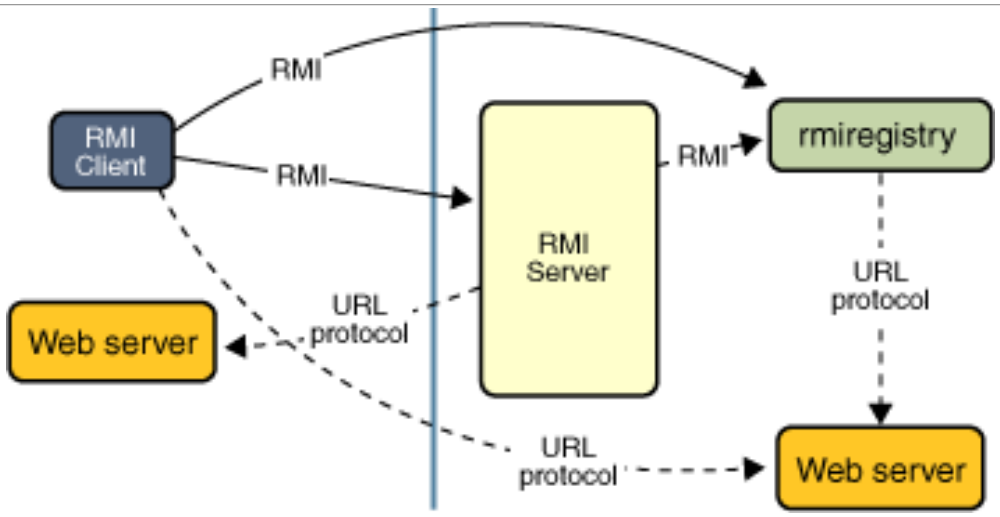
- Is RMI blocking on the client side?
  - YES: The client blocks until the server replies
- How can we make RMI non-blocking for the client's side?
  - Option 1: Make separate thread and let it block
  - Option 2: Or make the client also a Remote Object, pass it on to the server, and let it call you 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 need 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

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

- To set your custom security policy, start your application like this:

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

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