Remote Procedure Call

- Suited for Client-Server structure.
- Combines aspects of monitors and synchronous message passing:
  - ∗ Module (remote object) exports operations, invoked with `call`.
  - ∗ `call` blocks (delays caller) until serviced.
- `call` causes a new thread to be created on remote (server).
- Client-server synchronization and communication is implicit.

Terminology / Notation

server module: operations, (shared) variables, local procedures and threads for servicing remote procedure calls.

interface (specification): describes the operations, parameter types and return types.

`op opname(`*param types*`) [returns` *return type*`]`

server process: thread created by call to service an operation.

background process: threads running in a module that aren't created in response to call.

Example Server Module

```
module TicketServer
        op getNext returns int ;
body
        int next := 0 ;
        sem ex := 1 ;

        procedure getNext() returns val {
              P(ex) ;
              val := next ;
              next := next + 1 ;
              V(ex) ; }
end TicketServer
```

Issues

Lookup and registration

How does the client find the server?

Often server registers (binds) with a naming service (registry).

Client obtains information (lookup) about server from this server.

This changes the question to: *How does the client find the registry?*

Synchronization

Synchronization within a module (server). Two approaches:

1. Assume mutual exclusion in server (only one server process/background process executing at a time).
   - ∗ Similar to monitors.
   - ∗ Still need conditional synchronization.
2. Program it explicitly (i.e., using semaphores, monitors etc.).

Argument passing

Formats may be different on different machines.

- ints are different sizes, encodings, endianess.
- floats have different encodings

Address space is different in different processes.

- Can not pass pointers.
- Can not pass by reference.
- Can not pass objects containing pointers.

Three solutions:

- Copy-in: Arguments are converted to byte arrays (serialization) and reconstructed on the other side.
- Copy-in/copy-out: Copy-in + final value is passed back.
- Proxy objects: A proxy object is constructed on the server side. Calls to the proxy are converted to RPCs back to the argument.

Java RMI (Remote Method Invocation)

Client objects and server objects are local to different JVM processes.

Server objects (usually) extend java.rmi.server.UnicastRemoteObject.

Lookup and registration

How does the client find the server?

- Server objects registered by name with a registry service. (Naming.bind)
- Client objects obtain references to proxy objects from the registry service. (Naming.lookup)

Synchronization

Synchronization within a module (server).

- Each remote call implies the creation of a new server thread. So if there are multiple clients, there can be multiple server threads active at the same time.
- Synchronization must be programmed explicitly (use **synchronized** or my monitor package)

Argument passing

Formats may be different on different machines.

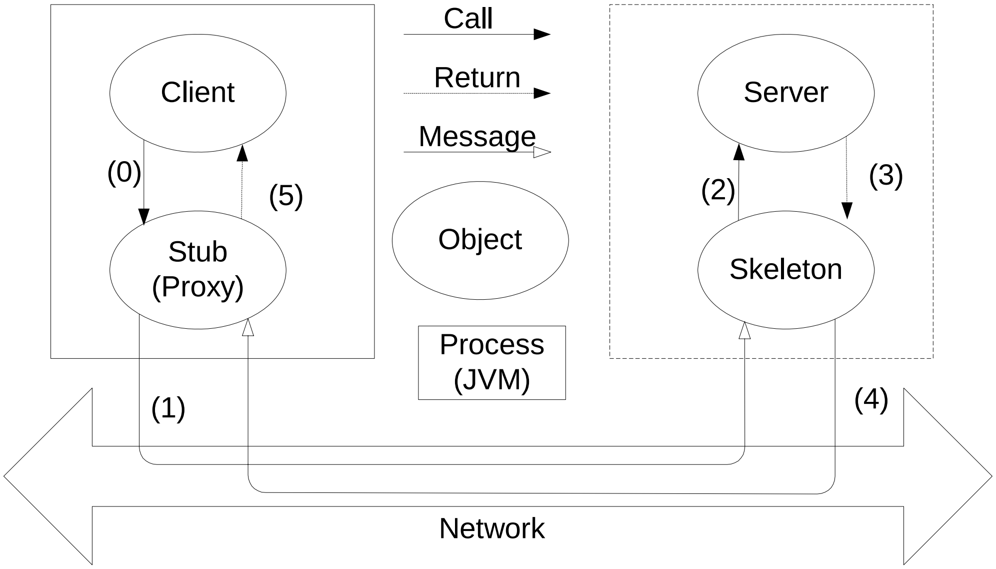
- Not an issue as data formats are standard across all Java implementations.

Address space is different in different processes.

- Reference arguments (and subsidiary references) are serialized and passed by copy-in rather than by reference.
- Except RemoteObjects, in which case a proxy (‘stub’) is passed instead (This complicates garbage collection).

Skeletons and Stubs

- 'Stub' objects implement the same interface as the server objects. (Proxy pattern)
- (0), (5) Client threads call a stub local to their JVM instance.
- (1), (4) Stub messages (TCP/IP) to Skeleton object in remote JVM & waits for reply.
- (2), (3) Skeleton creates a new server thread which calls the server.
- Stub and skeleton classes are synthesized by a RMI Compiler (rmic).



Example

Start a registry process

```
D:\...\classes> rmiregistry -Jcp -J.
```

First we need an interface for the server

```
package tryrmi;
import java.rmi.* ;

public interface TicketServerInterface extends Remote
{
    public int getTicket() throws RemoteException ; }
```

We implement the interface and write a main program to create and register the server object.

```
package tryrmi;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class TicketServer
extends UnicastRemoteObject
implements TicketServerInterface {

        private int next = 0 ;

        public synchronized int getTicket()
        throws RemoteException {
              return next++ ; }

        public TicketServer() throws RemoteException {
              super() ; }
```

```
    public static void main( String[] args ) {
        try {
            TicketServer server = new TicketServer()
           ;
            String name = args[0] ;
            Naming.bind( name, server ) ; }
        catch( java.net.MalformedURLException e) {
            ...  }
        catch( AlreadyBoundException e) { ... }
        catch( RemoteException e ) { ... } } }
```

Executing main creates and registers a server object.

```
D:\...\classes> java -cp . tryrmi.TicketServer
                                    rmi://frege.engr.mun.ca/ticket
```

The skeleton object will be generated automatically by the Java Runtime Environment. (There is no need to write a class for the skeleton object.)

Some client code

```
package tryrmi;
```

```
import java.rmi.* ;

public class TestClient {

    public static void main(String[] args) {
        try { String name = args[0] ;
            TicketServerInterface proxy =
                                    (TicketServerInterface)
                                  Naming.lookup( name ) ;
            use proxy object}
        catch( java.net.MalformedURLException e) {
            ... }
        catch( NotBoundException e) { ... }
        catch( RemoteException e) { ... } } }
```

The client can be executed from anywhere on the internet. (The stub object will be created by the Java Runtime Environment. There is no need to write a class for the stub object.)

```
D:\...\classes> java -cp . tryrmi.TestClient
                              rmi://frege.engr.mun.ca/ticket
```

RPC $\neq$ PC

Although remote procedure calls and local procedure calls are beguilingly similar and in Java share exactly the same syntax, there are important differences

- Partial Failure
  - ∗ Part of the system of objects may fail
  - ∗ Partial failures may be intermittent
  - ∗ Network delays
    - · On a large network, delays are indistinguishable from failures
- Performance
  - ∗ Remote calls are several orders of magnitude more expensive than local calls (100,000 or more to 1)
- Concurrency
  - ∗ Remote calls introduce concurrency that may not be in a nondistributed system.
- Semantics changes
  - ∗ In Java, local calls pass objects by reference.
  - ∗ Remote calls pass objects either by copy or by copying a proxy.