



T-106.420  
Concurrent Programming  
Remote Procedure Calls



# RPC and Rendezvous

- suited to client/server type interactions
- combine aspects of monitors and synchronous message passing
  - monitor aspect
    - operations exported
    - invoked by *call*
  - message passing aspect
    - *call* delays the caller (until results returned)
- what's new in RPC & Rendezvous
  - 2-way communication channel
  - synchronization



# RPC versus Rendezvous

- difference in *HOW* operations are serviced
- RPC
  - procedure for each operation
  - process for each call (conceptually)
- Rendezvous
  - done within an existing process
  - service by
    - input/accept statement (which waits for invocation)
    - process
    - return results



# module

```
module mname
  headers of exported operations;
body
  variable declarations;
  initialization code;
  procedures for exported operations;
  local procedures and processes;
end mname
```

- **specification and implementation part**



## operation header declaration

- these are "*exported*", i.e. can be called from processes of other modules  
meant for IPC (Inter Process Communication)

**op** **opname** (formals) [**returns** result]

- `formals` and `result` specify the types and optionally the names



## Body procedures, implementation

```
proc opname (formal identifiers) returns result identifier  
  declarations of local variables;  
  statements  
end
```

- types of parameters and result can be omitted (already specified in the declaration)
- called *proc*, because a call of it creates a new process (at least conceptually (see before))



# invocation

- process (or procedure) calls a procedure in (*another*) module

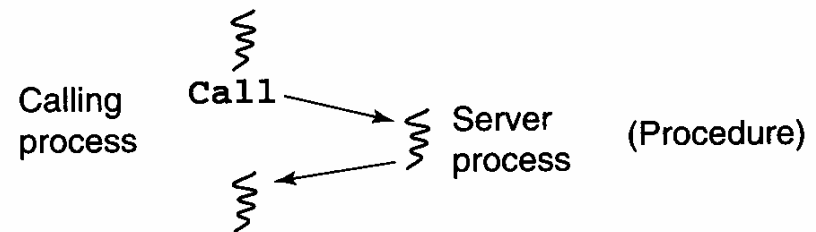
**call mname . opname** (arguments)

- omissions
  - **call** , if function call
  - **mname** , if local call



# Invocation (2)

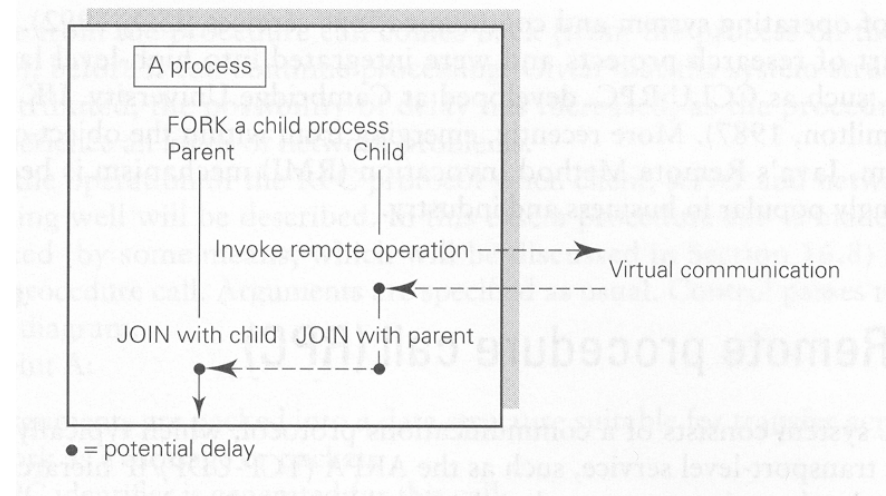
- intermodule call
  - address spaces different (probably)
  - new process services the call (created or allocated)
    - args passed as messages
    - result passed as messages
- local call
  - same address space
  - caller process can execute procedure body



# implementation of remote call forking a child

to make synchronous call

- child will make remote invocation and WAIT for the result and then
- synchronize with parent
- facility for *distributed sequential programming*
- facility for *concurrency* is the *FORK*ing





# synchronization

## assumptions

- all processes within a module execute with *mutual exclusion*
  - *mutual exclusion* is implicit (analogous to monitors)
  - (condition synchronization) implemented with
    - condition variables
    - semaphores
  - simpler, easier (context switch only at entry, exit, or delay points)
- processes execute *concurrently* (our assumption)
  - *mutual exclusion* has to be explicitly programmed
  - (condition synchronization) implemented with
    - semaphores
    - local monitors
    - rendezvous
  - ”runaway” processes can be handled (if e.g. time slicing)



# a time server

- **get\_time, wake\_time**  
safe to execute  
concurrently, because  
only read **tod**
- **delay & Clock** mutually  
exclusive for **napQ**  
access

```
module TimeServer
  op get_time() returns int; # retrieve time of day
  op delay(int interval);   # delay interval ticks
body
  int tod = 0;              # the time of day
  sem m = 1;                # mutual exclusion semaphore
  sem d[n] = ([n] 0);      # private delay semaphores
  queue of (int waketime, int process_id) napQ;
  ## when m == 1, tod < waketime for delayed processes

  proc get_time() returns time {
    time = tod;
  }

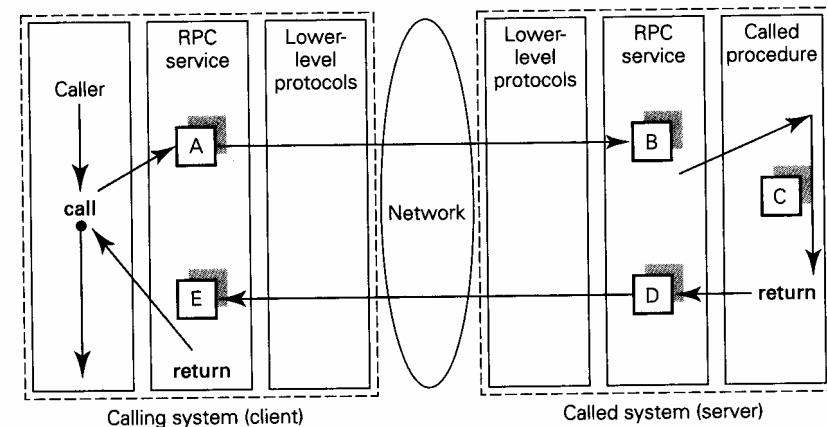
  proc delay(interval) {    # assume interval > 0
    int waketime = tod + interval;
    P(m);
    insert (waketime, myid) at appropriate place on napQ;
    V(m);
    P(d[myid]); # wait to be awakened
  }

  process Clock {
    start hardware timer;
    while (true) {
      wait for interrupt, then restart hardware timer;
      tod = tod+1;
      P(m);
      while (tod >= smallest waketime on napQ) {
        remove (waketime, id) from napQ;
        V(d[id]); # awaken process id
      }
      V(m);
    }
  }
end TimeServer
```

# an RPC system

## elements

- communications protocol (on top of transport layer)
- language-level routines for data assembly
- mechanism to bind *remote procedure* names to network addresses





## an RPC system (2)

- the calling process has to wait till the response from the procedure call comes back
- at A
  - arguments packed for transfer as packet
  - RPC identifier generated for the call
  - timer set
- data passed to lower level protocols
- at B
  - arguments unpacked for making *local* procedure call
  - RPC identifier is noted



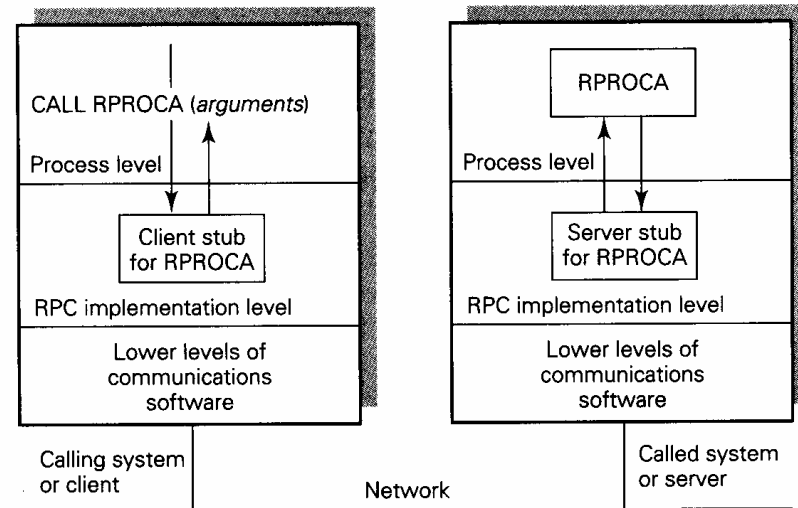
## an RPC system (3)

- at C
  - remote procedure executed
- at D
  - return arguments packed for transfer as packet
  - timer set
- data passed to lower level protocols
- at E
  - return arguments unpacked for making *local* procedure call
  - timer set at A is disabled
  - acknowledgement sent for this RPC identifier (timer set at D will be disabled)



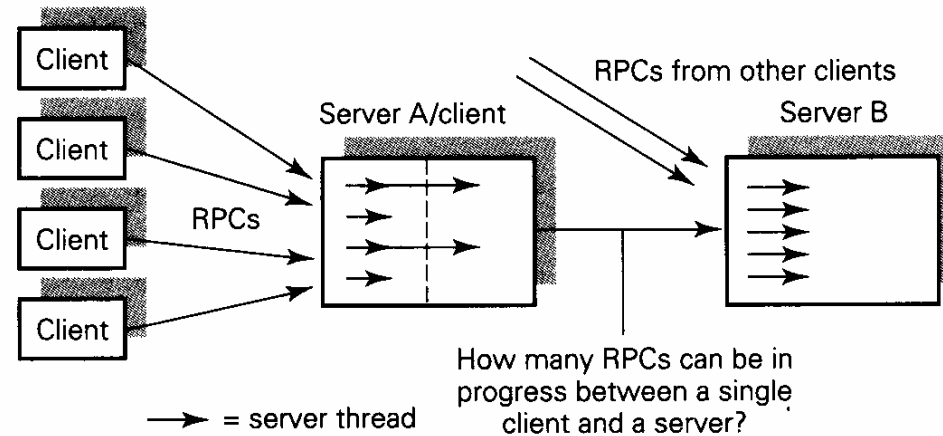
# RPC language integration

- into a language that uses procedure calls
- programmer freed of distribution detail issues
- transparent approach: creates stub/proxy with same name as remote
- reference parameter and return argument issues





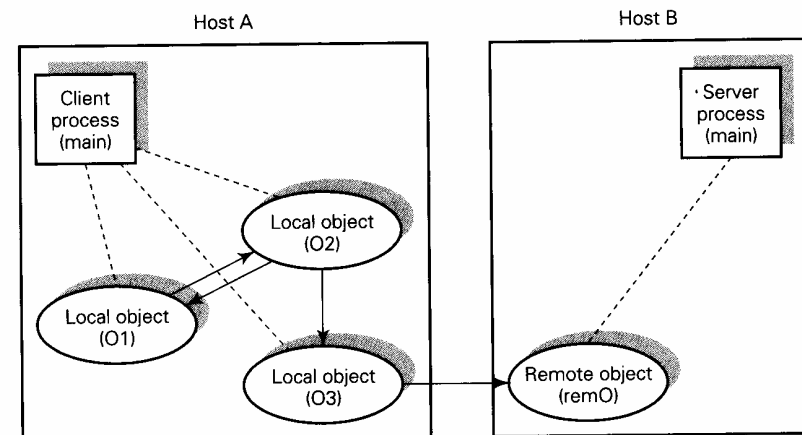
# clients, servers and RPC's





# Java RMI

- Remote Method Invocation
- `java.rmi` package
- dotted lines: association with process
- arrow lines: method invocation
- data flow also in other direction

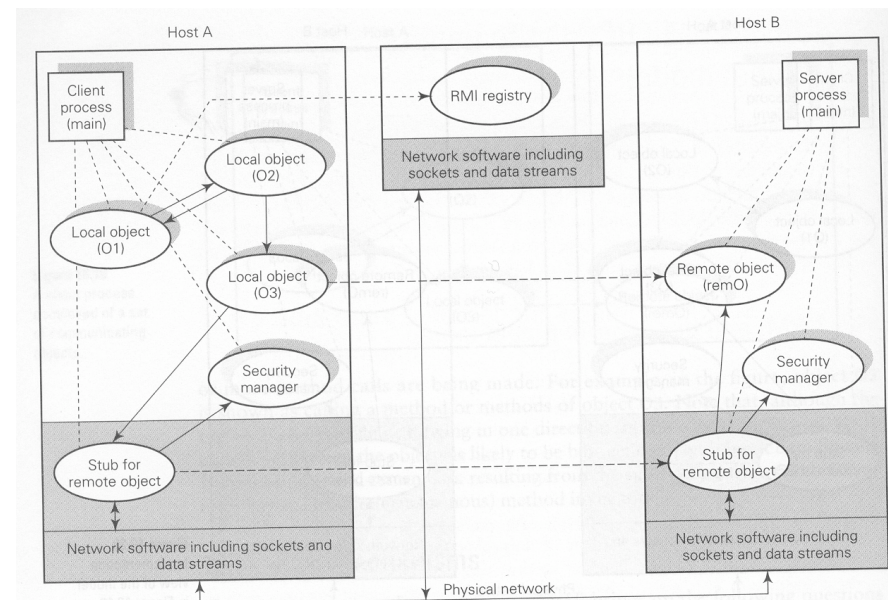




# Java RMI (2)

## RMI mechanisms

- remote object creation and naming
- how clients learn name & location of remote objects
- invocation from local to remote object





# remote object creation and naming

before a remote object can be created

- its operations must be specified in an interface which extends interface **Remote**
- the remote object's class (**InterestRateImpl**) must be defined by implementing this interface and extending the class **UnicastRemoteObject**

```
import java.rmi.*;
public interface InterestRate extends Remote{
    public void setCurrentRate (double sr) throws RemoteException;
    public double getCurrentRate() throws RemoteException;
}
```

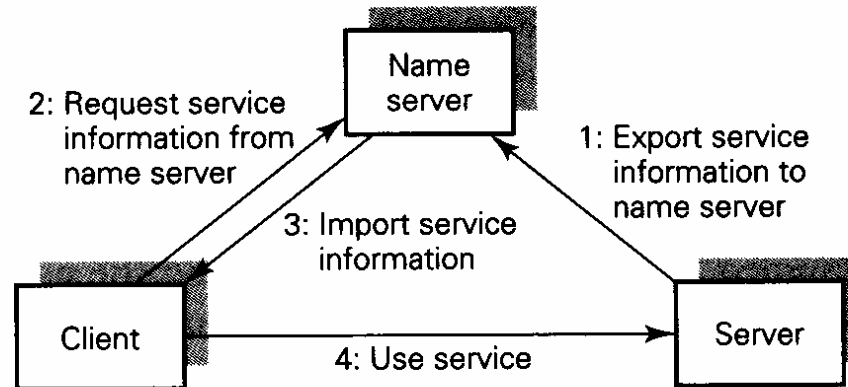
Figure 16.22  
The interface for  
class InterestRate.

```
import java.rmi.*;
import java.rmi.server.*;
public class InterestRateImpl extends UnicastRemoteObject implements InterestRate {
    double currentRate = 0;
    public InterestRateImpl(double cr) throws RemoteException{
        currentRate = cr;
    }
    public void setCurrentRate (double sr) throws RemoteException {
        currentRate = sr;
    }
    public double getCurrentRate () throws RemoteException{
        return currentRate;
    }
}
```

Figure 16.23  
The implementation  
for Remote Object  
InterestRate.



# name server in a distributed system





# how clients learn name & location of remote objects

before a remote object can be created

- declare & create object of (remote) class (**InterestRateImpl**)
- register it in **RMI registry** using a string name as an identifier
- clients find remote objects from RMI registry knowing
  - location of registry
  - the string name of object (**CurrentInterestRate**)

Figure 16.24  
The server registers the remote object with the RMI registry.

```
import java.rmi.*;
public class InterestRateServer {
    public static void main(String[] args) {
        InterestRateServer irs = new InterestRateServer();
    }
    public InterestRateServer() {
        try {
            InterestRate ir = new InterestRateImpl(5.5);
            Naming.rebind("rmi://aServerName/CurrentInterestRate", ir);
            System.out.println("Remote object registered");
        } catch (Exception e) {System.out.println("Trouble: " + e);}
    }
}
```

Figure 16.25  
Client can look up remote object in RMI registry, and use it.

```
import java.rmi.*;
public class InterestRateClient {
    public InterestRate ir;
    public InterestRateClient() {
        try{
            ir = (InterestRate)
                Naming.lookup("rmi://aServerName/CurrentInterestRate");
            System.out.println("Current Interest Rate = " + ir.getCurrentRate());
        }
        catch (Exception e) {System.out.println(e);}
    }
    public static void main(String[] args) {
        InterestRateClient irc = new InterestRateClient();
    }
}
```



# invocation from local to remote object

- RMI registry returns a referent to a **stub** object
- a copy of it is placed both on client and server machine
- client method invocation works on client stub
  - client stub communicates with *similar* stub on remote
  - server stub makes call to the remote object itself
- stubs (un)marshal arguments of remote call and the return value