ID1217 Concurrent Programming
Lecture 17

ROYAL INSTITUTE OF TECHNOLOGY

Remote Procedure Call (RPC); Rendezvous; Remote Method Invocation (RMI). Java RMI

Vladimir Vlassov

KTH/ICT/EECS

Outline

- Remote Procedure Call (RPC)
  - Syntax of RPC
  - Executing a remote procedure call
- Rendezvous
  - Input statements (accept, select)
  - Typical rendezvous
- Examples of RPC and rendezvous
- RPC versus Rendezvous
- Multiple primitives notation
- Remote Method Invocation (RMI)
- Case study: Java RMI

Java RMI References

- Trail: RMI
  http://docs.oracle.com/javase/tutorial/rmi/index.html

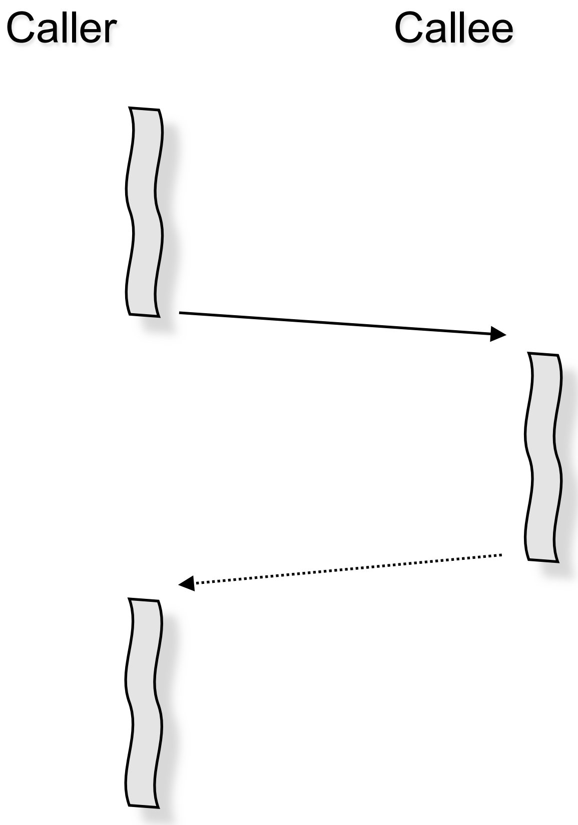
- Java RMI
  http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html

Motivation for RPC and RMI

- **Message passing** is convenient for consumers-producers (filters) and P2P, but it is somewhat low level for client-server applications
  - Client/server interactions are based on a request/response protocol;
  - Client requests are typically mapped to procedures on server;
  - A client waits for a response from the server.
- Need for more convenient (easier to use) communication mechanisms for developing client/server applications
- **Remote Procedure Call (RPC) and rendezvous**
  - Procedure interface; message passing implementation
- **Remote Method Invocation (RMI)**
  - RMI is an object-oriented analog of RPC
- RPC, rendezvous and RMI are implemented on top of message passing.

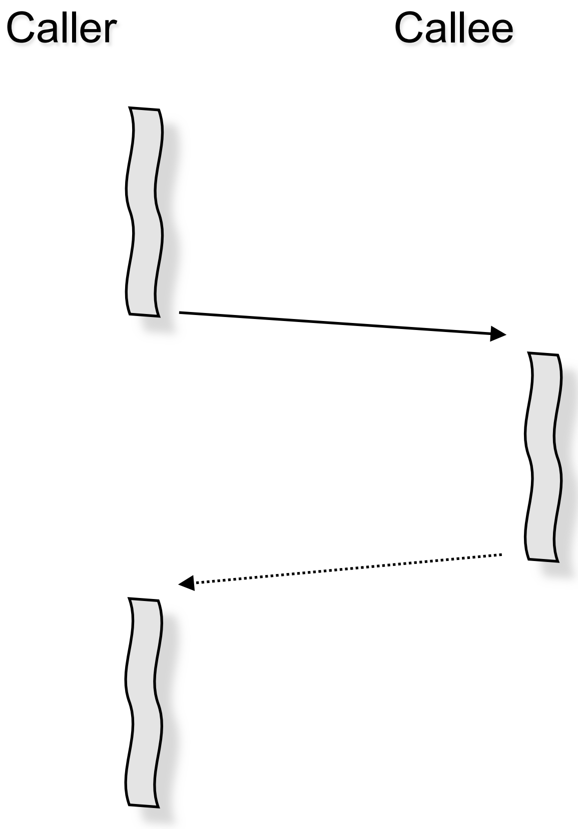
RPC: Remote Procedure Call

- **RPC** is a mechanism that allows a program running on one computer (VM) to cause a procedure to be executed on another computer (VM) without the programmer needing to explicitly code for this.
- Two processes involved: **caller** and **callee**.
- **Caller (RPC client)** is a **calling process** that initiates an RPC to a server.
- **Callee (RPC server)** is a **called process** that accepts the call.



RPC: Remote Procedure Call (cont’d)

- Each RPC is executed in a **separate process (thread)** on the server side
- **An RPC is a synchronous operation.**
  - The caller is suspended until the results of the remote procedure are returned.
  - Like a regular or local procedure call.
  - Guess why?

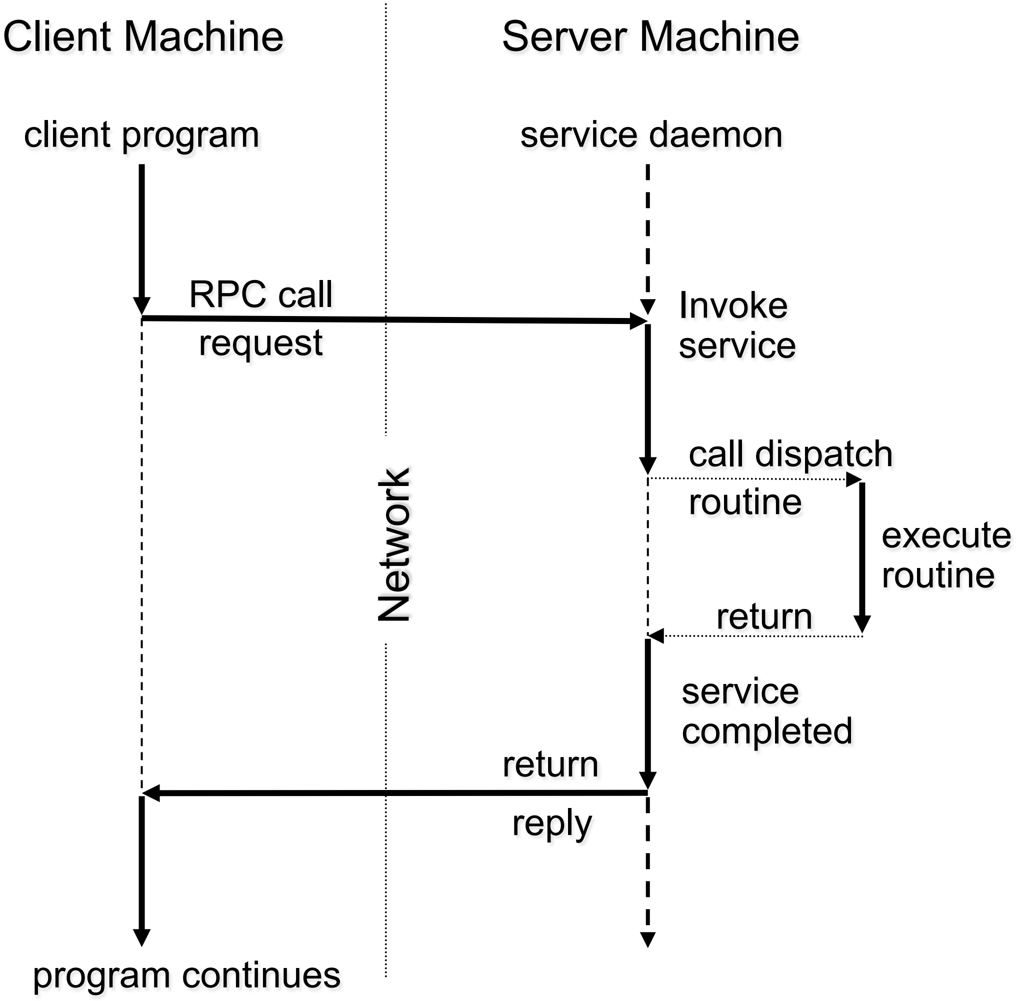


Identifying a Remote Procedure

- Each RPC procedure is **uniquely identified** by
  - A program number
    - identifies a group of related remote procedures
  - A version number
  - A procedure number
- An RPC call message has **three unsigned fields**:
  - Remote program number
  - Remote program version number
  - Remote procedure number
  - The three fields uniquely identify the procedure to be called.

Executing RPC

- On each RPC the server starts a **new process** to execute the call.
- The new process terminates when the procedure returns and results are sent to the caller.
- Calls from the same caller and calls from different callers are serviced by **different concurrent processes** on server.
- Concurrent invocations might interfere with each other when accessing shared objects – **might need synchronization**



RPC Syntax

- Modules (Servers)

```
module mname
     interface, i.e. headers of exported operations;
body
      variable declarations;
      initialization code;
      procedures for exported operations;
      local procedures and processes;
end mname
```

- Exported operation (method of a remote interface)

```
op opname(formal identifiers) [returns result]
```

- Procedure – operation implementation

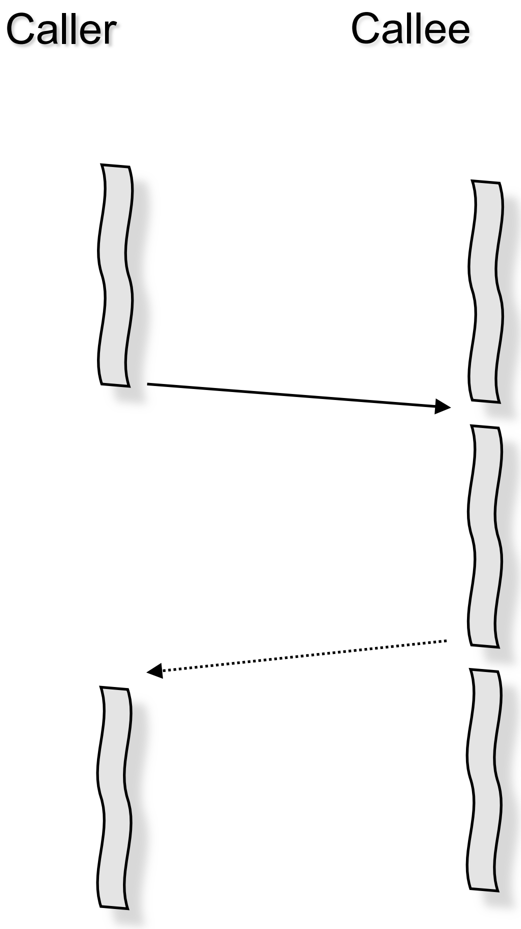
```
proc opname(formal identifiers) returns result identifier
      declarations of local variables;
      statements
end
```

- Client makes a remote call to a module (server):

```
call mname.opname(arguments)
```

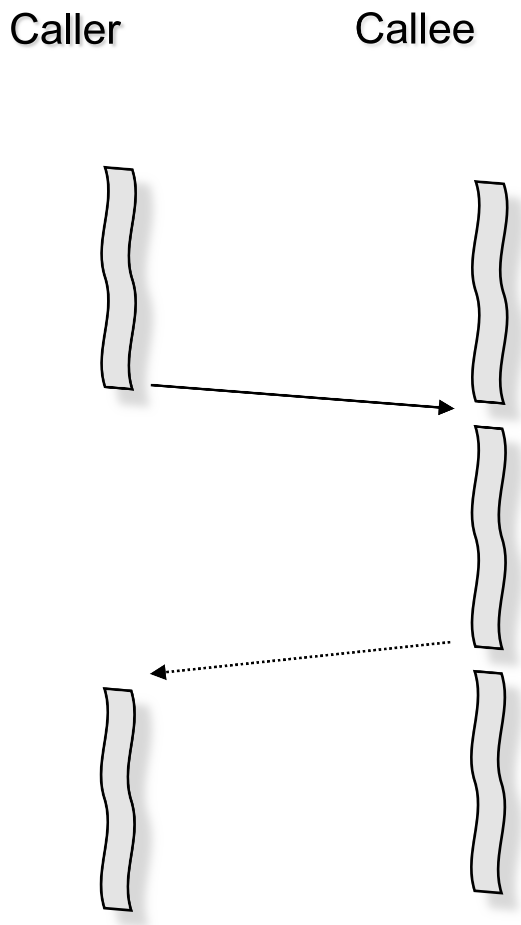
Rendezvous

- **Rendezvous** – remote invocation on a running process
- The server proc is not always available, and execution of a call might be deferred on the server until the server **accepts** the call
- Central inter-process communication mechanism in ADA



Properties of Rendezvous

- Each remote call is serviced by **the same server process**
- As only one servicing process is active, so avoiding interference.
- In either case, RPC or Rendezvous, **client blocks until remote procedure completes** and result is returned (received).



Rendezvous Syntax (similar to ADA)

- Declaration of operations (**server interface**):

```
op opname (types of formals);
```

- **Input statement:**
  - The simplest form (**accept**) defines one operation that can be called

```
in opname (formal identifiers) [returns result] -> S; ni
```

  - The general form (**select**) defines a list of guarded operations

```
in opname_1 (formals_1) and B_1 by e_1 -> S_1;
[] …
[] opname_n (formals_n) and B_2 by e_n -> S_n;
ni
```

- **Call** of a remote operation by client – like in RPC

```
call mname.opname (arguments)
```

The Input Statement

```
in opname_1 (formals_1) and B_1 by e_1 -> S_1;
[] …
[] opname_n (formals_n) and B_2 by e_n -> S_n;
ni
```

- The part before **->** is a **guard** of S_i
  - B_i – **optional synchronization expression;**
  - e_i – **optional scheduling expression;**
  - Synchronization and scheduling expressions may reference formals.
- Execution rules:
  - Execution of the input statement delays until some guard succeeds;
  - A guard of **op** succeeds when the **op** has been called and **B** is true (or omitted);
  - If more then one guard succeeds, the oldest one is selected if **e** is omitted;
  - **e** controls selection: an **op** with minimum value of **e** is selected for execution
  - The statement terminates, when execution of the selected **op** completes and results are sent to the caller.

Some Typical Rendezvous

- Basic input statement defines a remote subroutine (similar to **accept** in ADA)

```
in opname(formals) -> body ni
```

- Multiple entries (similar to **select** in ADA)

```
in readblk(...) -> ...
[] writeblk(...) -> ...
ni
```

- Delay based on local state using synchronization expression:

```
in request() and avail > 0 -> avail--; ...
[] release() -> avail++; ...
ni
```

- Delay based on arguments and local state:

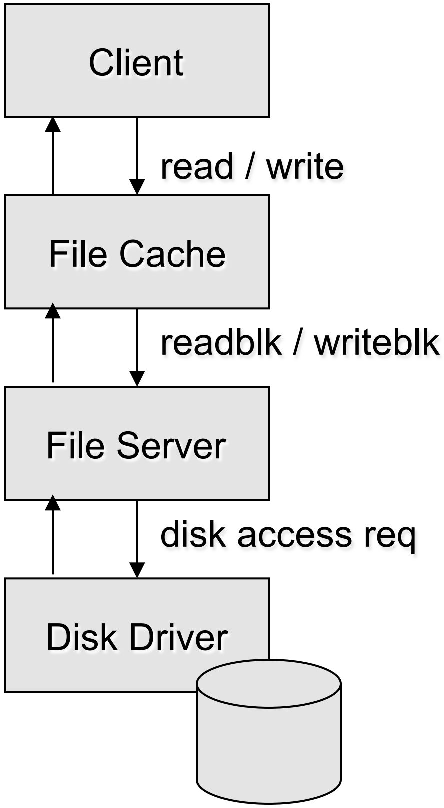
```
in getforks(i) and not eating[left(i)] and not eating[right(i)] -> eating[i]
   = true;
[] relforks(i) -> eating[i] = false;
ni
```

- Scheduling based on arguments using scheduling expression (by)

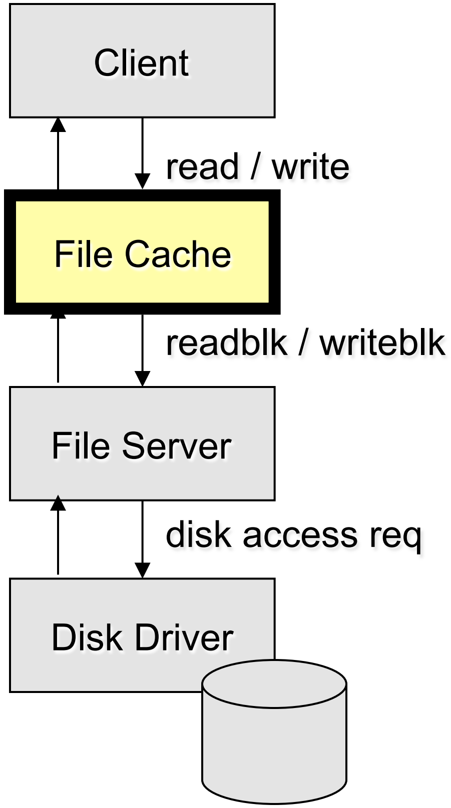
```
in request(time) and free by time -> free := false;
[] release() -> free := true;
ni
```

Example: A Distributed File System Using RPC

- Modules (servers):
  - **FileCache**
    - A write-back allocate-on-write cache of file blocks
    - Exports **read** and **write** operations
    - On a miss calls remote **FileServer**
  - **FileServer**
    - Provides access to file blocks stored on a disk
    - Exports **readblk** and **writeblk**
    - Uses the local **DiskDriver** process to access the disk.

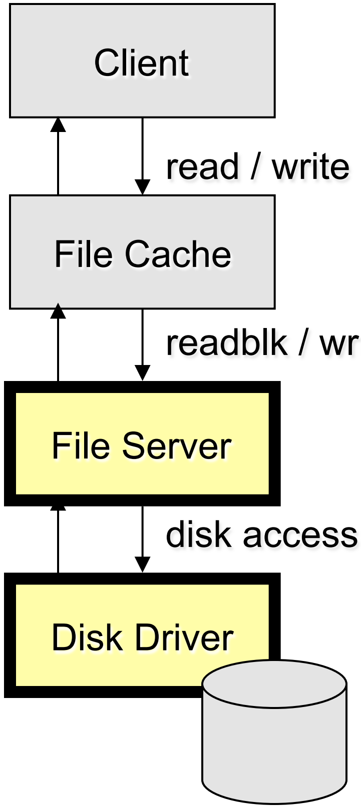


The File Cache



```
module FileCache   # located on each diskless workstation
  op read(int count; result char buffer[*]);
  op write(int count; char buffer[]);
body
  cache of file blocks;
  variables to record file descriptor information;
  semaphores for synchronization of cache access (if needed);

  proc read(count,buffer) {
    if (needed data is not in cache) {
      select cache block to use;
      if (need to write out the cache block)
        FileServer.writeblk(...);
      FileServer.readblk(...);
    }
    buffer = appropriate count bytes from cache block;
  }

  proc write(count,buffer) {
    if (appropriate block not in cache) {
      select cache block to use;
      if (need to write out the cache block)
        FileServer.writeblk(...);
    }
    cache block = count bytes from buffer;
  }
end FileCache
```

The File Server and The Disk Driver



Client

read / write

File Cache

readblk / writeblk

File Server

disk access req

Disk Driver

```
module FileServer   # located on a file server
  op readblk(int fileid, offset; result char blk[1024]);
  op writeblk(int fileid, offset; char blk[1024]);
body
  cache of disk blocks;
  queue of pending disk access requests;
  semaphores to synchronize access to the cache and queue;
  # N.B. synchronization code not shown below

  proc readblk(fileid, offset, blk) {
    if (needed block not in the cache) {
      store read request in disk queue;
      wait for read operation to be processed;
    }
    blk = appropriate disk block;
  }

  proc writeblk(fileid, offset, blk) {
    select block from cache;
    if (need to write out the selected block) {
      store write request in disk queue;
      wait for block to be written to disk;
    }
    cache block = blk;
  }

  process DiskDriver {
    while (true) {
      wait for a disk access request;
      start a disk operation; wait for interrupt;
      awaken process waiting for this request to complete;
    }
  }
end FileServer
```

RPC versus Rendezvous

- In both, RPC and rendezvous
  - A client issues remote calls on a server.
  - Each call is **synchronous**: the client blocks until remote procedure completes and result is returned (received).
- RPC:
  - On each remote call, a server creates a **new process** to handle the call.
  - A server should be always available.
  - As the calls as executed in separate threads – **synchronization** might be needed to avoid interference between calls (due to shared resources).
- Rendezvous:
  - Calls are accepted, selected and served by an **existing process** – no interference.
  - Inter-process communication mechanism in ADA.
- Both, RPC and rendezvous are ideally suited for programming client/server applications (interactions).

Multiple Primitives Environment

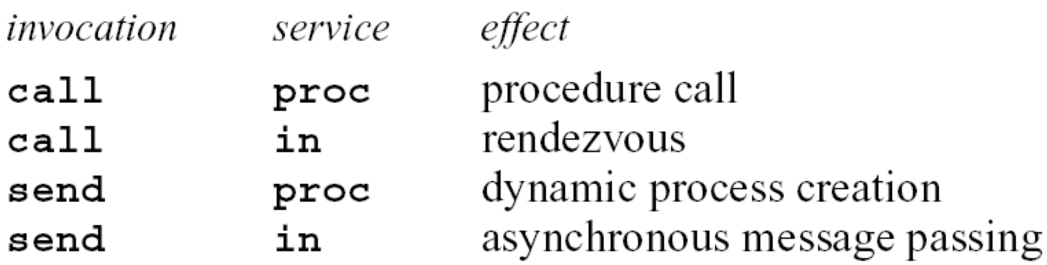
- Combines RPC, rendezvous and message passing in one coherent programming environment.
- A server exports operations that can be executed on a client request
  - In a new process:

    ```
    proc op(formals) { … }
    ```

  - In the server process that executes an input statement:

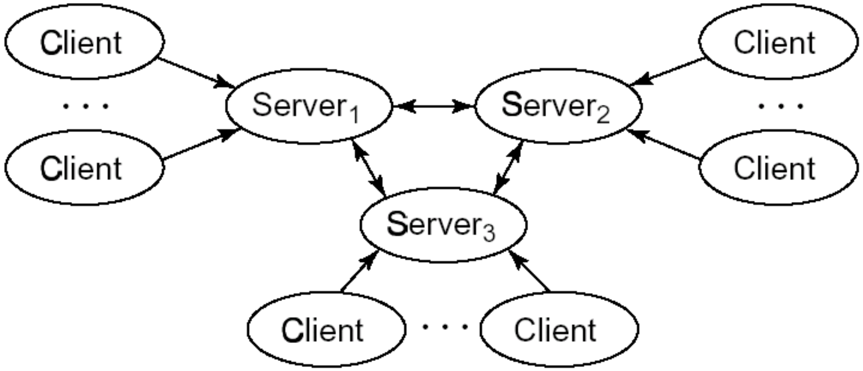
    ```
    in op1(formals1) -> S1; [] op2(formals2) -> S2;…ni
    ```

- Client interacts with a server's module using:
  - Either RPC: `call Mname.opname(`arguments`);`
  - Or message passing: `send Mname.opname(`arguments`);`
- Four possible combinations:



invocation	*service*	*effect*
`call`	`proc`	procedure call
`call`	`in`	rendezvous
`send`	`proc`	dynamic process creation
`send`	`in`	asynchronous message passing

Example: Replicated Files

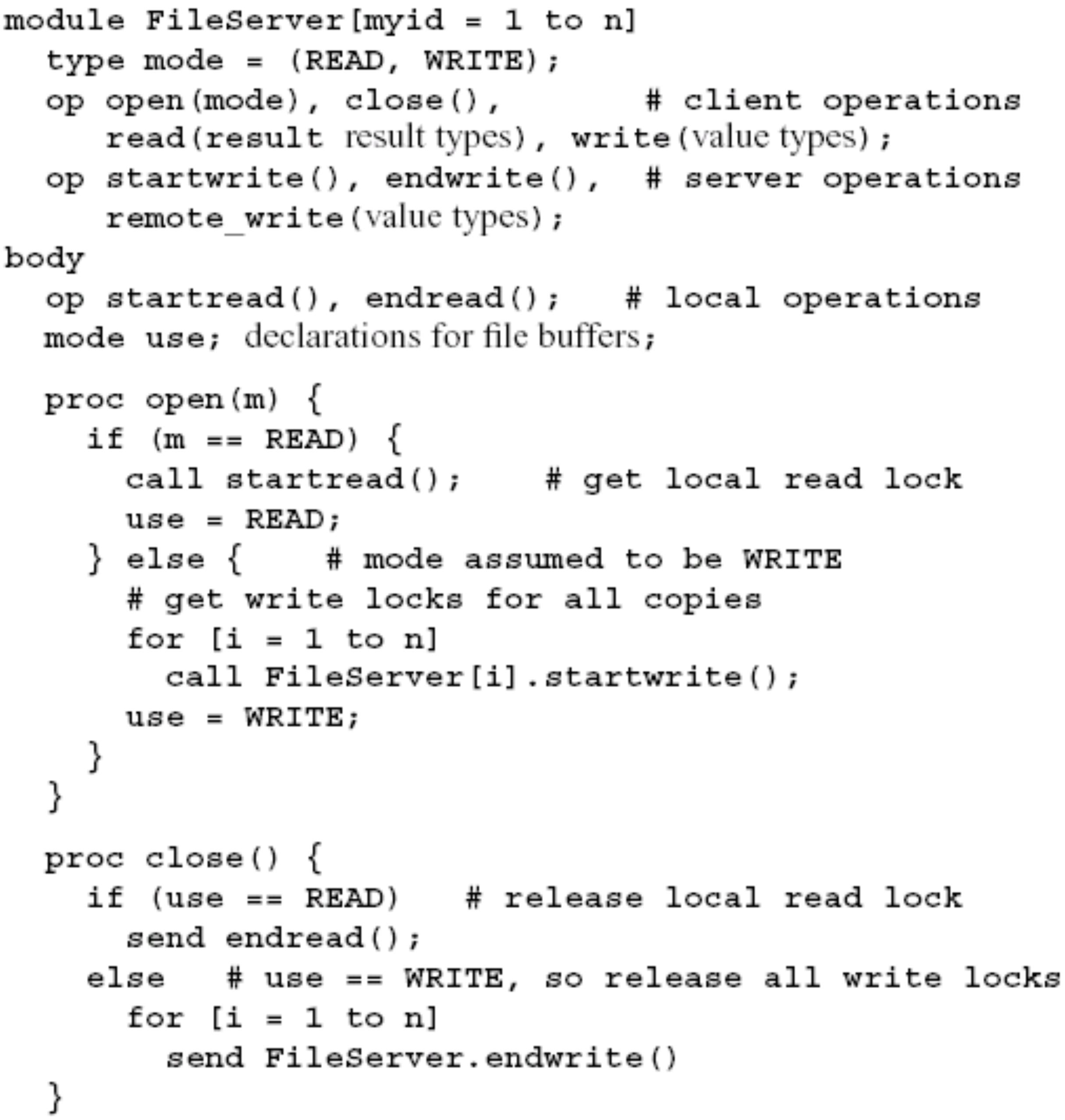


- Suppose:
  - **n** distributed file servers, each exports **open**, **close**, **read** and **write** operations.
  - A file is replicated on each server.
  - A client knows and communicates with one (nearest) server.
  - Servers use a write-update protocol to maintain coherency of copies.
- Invariant:

  `(nr == 0 ∨ nw == 0) ∧ nw ≤ 1`

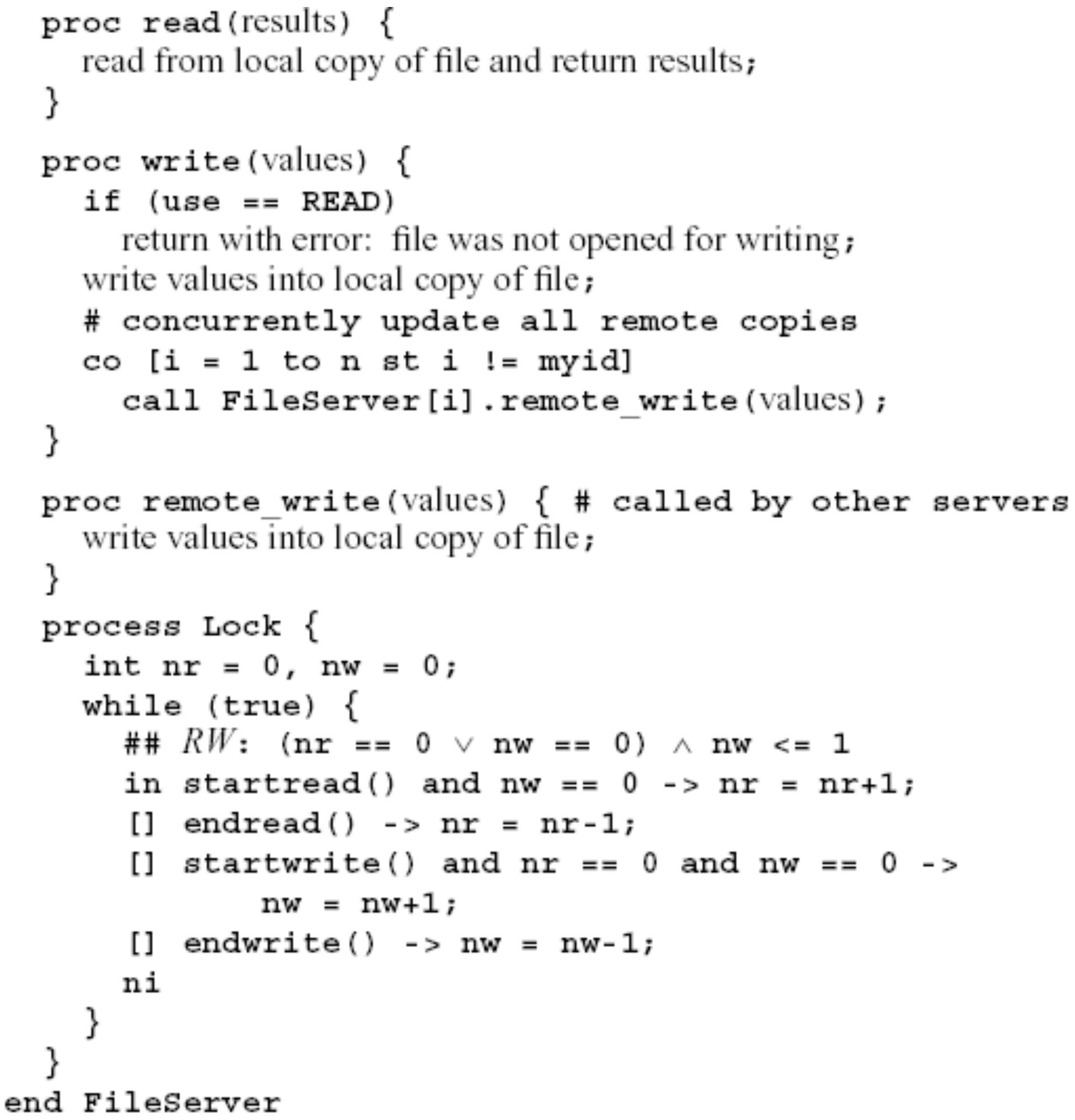
  - Concurrent reads, exclusive writes.
  - Assume one lock per copy.
  - When a file is open for reading, a local copy of the file must be locked.
  - When the file is open for writing, all replicas of the file must be locked.

The File Server



```
module FileServer[myid = 1 to n]
  type mode = (READ, WRITE);
  op open(mode), close(),         # client operations
     read(result  result types), write(value types);
  op startwrite(), endwrite(),   # server operations
     remote_write(value types);
body
  op startread(), endread();    # local operations
  mode use; declarations for file buffers;

  proc open(m) {
    if (m == READ) {
      call startread();     # get local read lock
      use = READ;
    } else {     # mode assumed to be WRITE
      # get write locks for all copies
      for [i = 1 to n]
        call FileServer[i].startwrite();
      use = WRITE;
    }
  }

  proc close() {
    if (use == READ)    # release local read lock
      send endread();
    else    # use == WRITE, so release all write locks
      for [i = 1 to n]
        send FileServer.endwrite()
  }
```

File Server (cont'd)

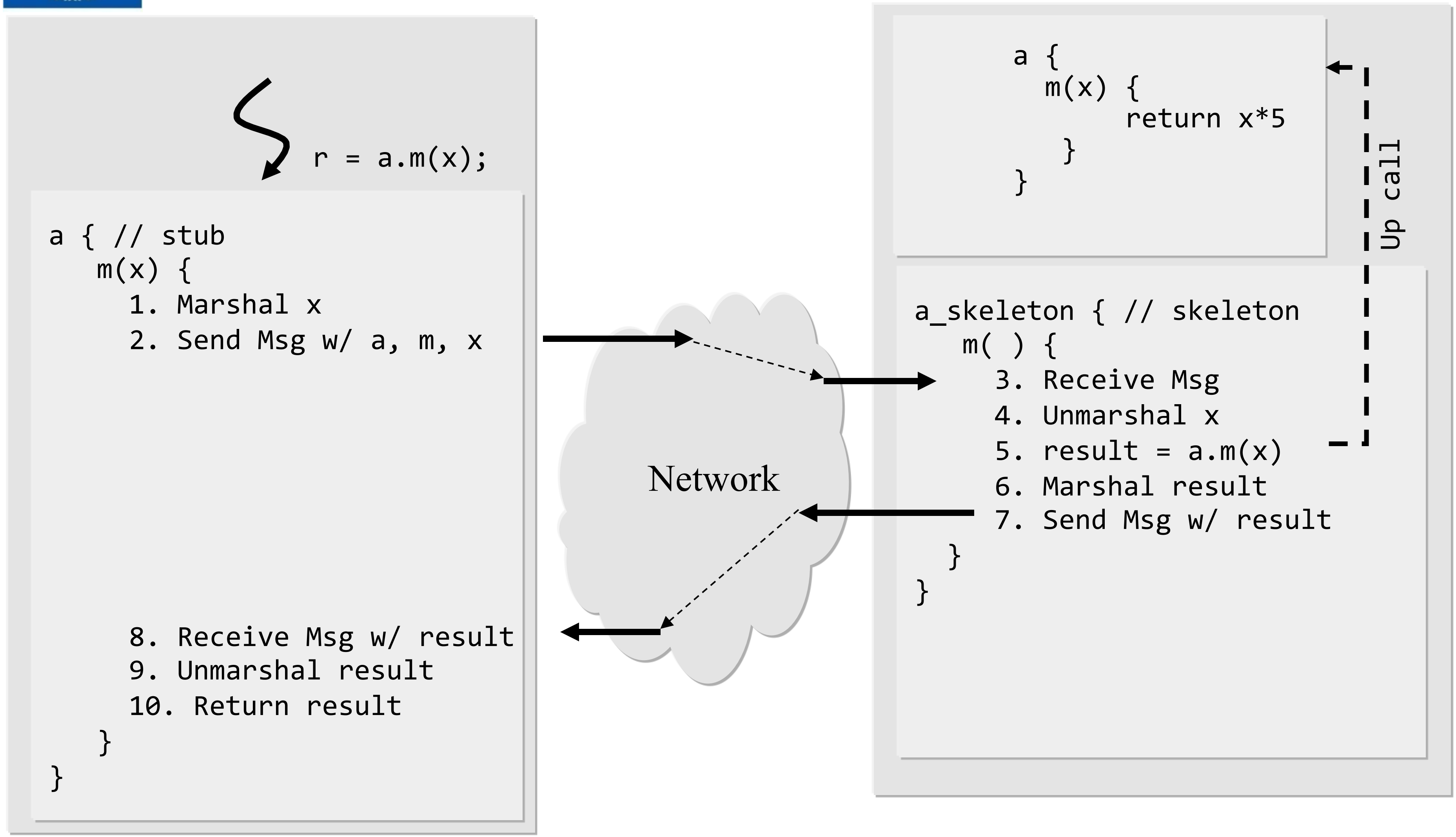


```
  proc read(results) {
    read from local copy of file and return results;
  }

  proc write(values) {
    if (use == READ)
      return with error:  file was not opened for writing;
    write values into local copy of file;
    # concurrently update all remote copies
    co [i = 1 to n st i != myid]
      call FileServer[i].remote_write(values);
  }

  proc remote_write(values) { # called by other servers
    write values into local copy of file;
  }
  process Lock {
    int nr = 0, nw = 0;
    while (true) {
      ## RW:  (nr == 0 ∨ nw == 0) ∧ nw <= 1
      in startread() and nw == 0 -> nr = nr+1;
      [] endread() -> nr = nr-1;
      [] startwrite() and nr == 0 and nw == 0 ->
             nw = nw+1;
      [] endwrite() -> nw = nw-1;
      ni
    }
  }
end FileServer
```

Remote Method Invocation (RMI)

- **Remote method invocation (RMI)** is a mechanism to invoke a method on remote object, i.e. object in another computer or virtual machine.
- RMI is the object-oriented analog of RPC in an distributed OO environment, e.g. OMG CORBA, Java RMI, .NET
  - RPC allows calling procedures over a network
  - RMI invokes objects' methods over a network
- **Location transparency**: invoke a method on a stub like on a local object
- **Location awareness**: the stub makes remote call across a network and returns a results via stack

Remote Method Invocation



```
r = a.m(x);
```

```
a { // stub
   m(x) {
     1. Marshal x
     2. Send Msg w/ a, m, x




     8. Receive Msg w/ result
     9. Unmarshal result
     10. Return result
   }
}
```

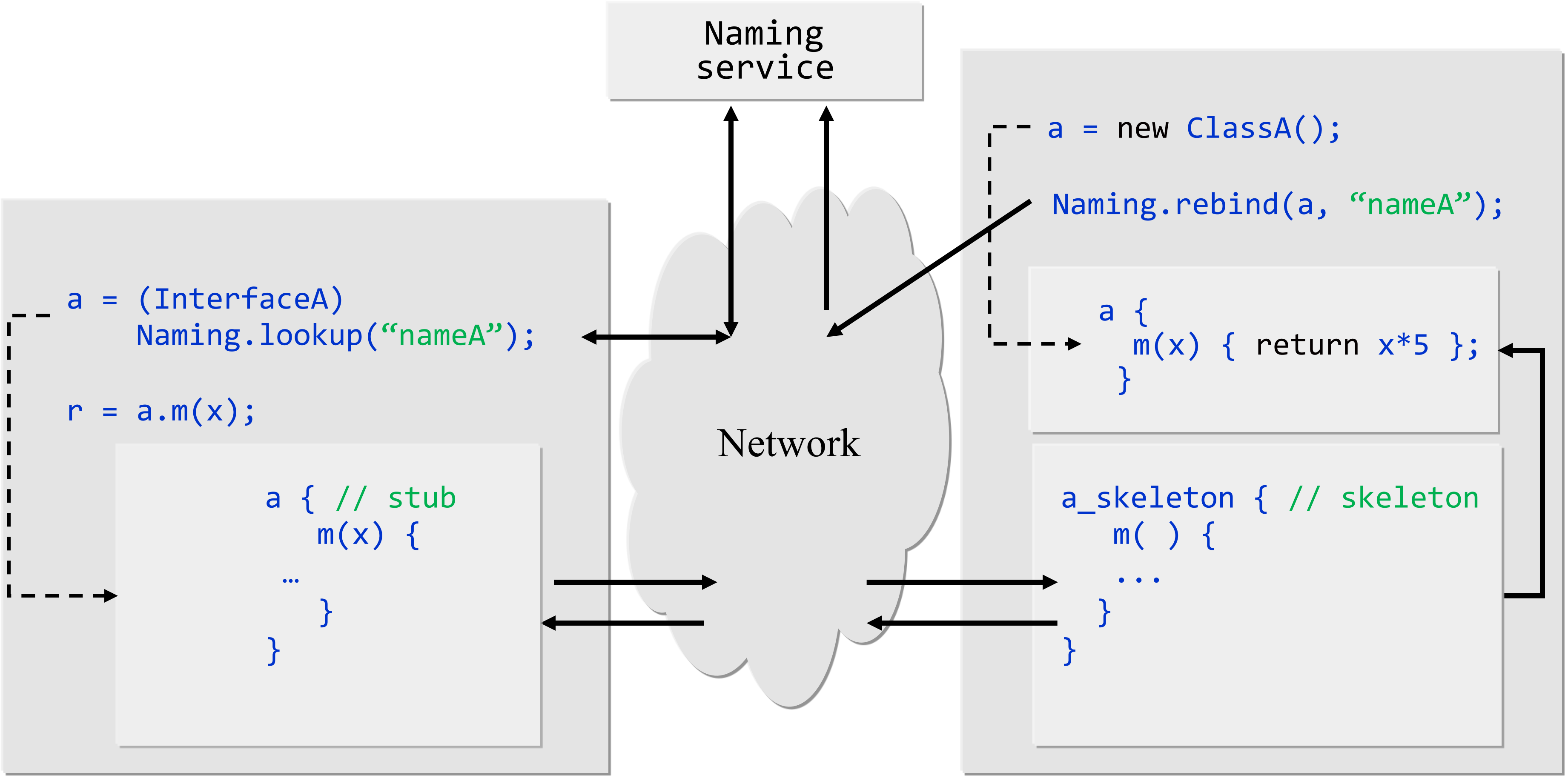
```
a {
  m(x) {
       return x*5
   }
}
```

```
a_skeleton { // skeleton
   m( ) {
     3. Receive Msg
     4. Unmarshal x
     5. result = a.m(x)
     6. Marshal result
     7. Send Msg w/ result
   }
}
```

Locating Objects

- How does a caller get a reference to a remote object, i.e. stub?
- One approach is to use a distributed **Naming Service**:
  - Associate a unique name with an object.
  - Bind the name to the object at the Naming Service.
    - The record typically includes name, class name, object reference (i.e. location information) and other information to create a stub.
  - The client looks up the object by name in the Naming Service.
- The primary reference problem: How to locate the Naming Service?
  - Configuration problem: URL of the naming service

Use of Naming Service



Naming service

Network

```
a = (InterfaceA)
    Naming.lookup("nameA");

r = a.m(x);
```

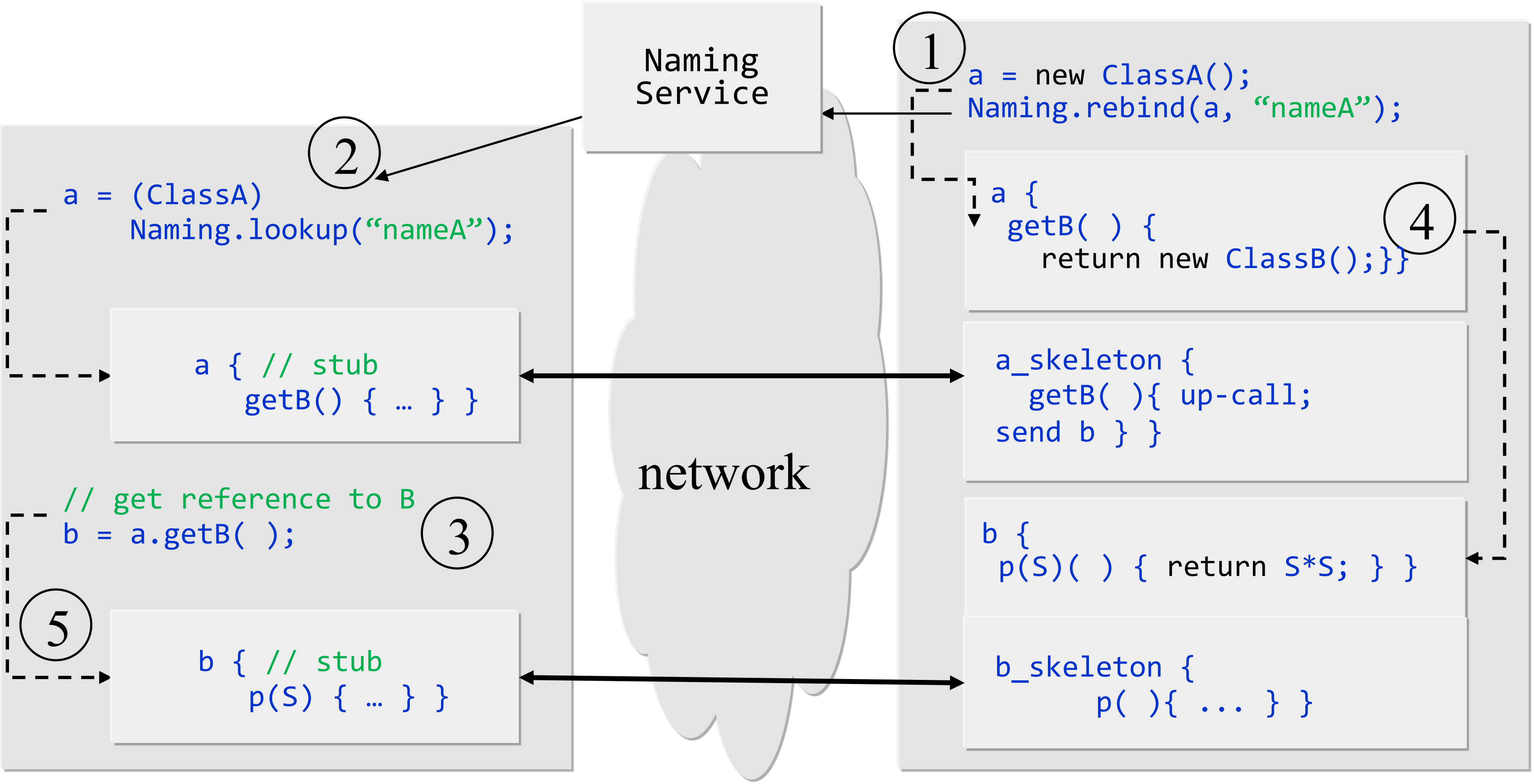
```
a { // stub
   m(x) {
  …
   }
}
```

```
a = new ClassA();

Naming.rebind(a, "nameA");
```

```
a {
  m(x) { return x*5 };
}
```

```
a_skeleton { // skeleton
   m( ) {
   ...
   }
}
```

Remote Reference in Return



Case Study: Java RMI

Java RMI (Remote Method Invocation)

- **Java RMI** is a mechanism that allows a thread in one JVM to invoke a method on a object located in another JVM.
  - Provides Java native ORB (Object Request Broker)
- The Java RMI facility allows applications or applets running on different JVMs, to interact with each other by invoking remote methods:
  - Remote reference (stub) is treated as local object.
  - Method invocation on the reference causes the method to be executed on the remote JVM.
  - Serialized arguments and return values are passed over network connections.
  - Uses Object streams to pass objects “by value”.

RMI Classes and Interfaces

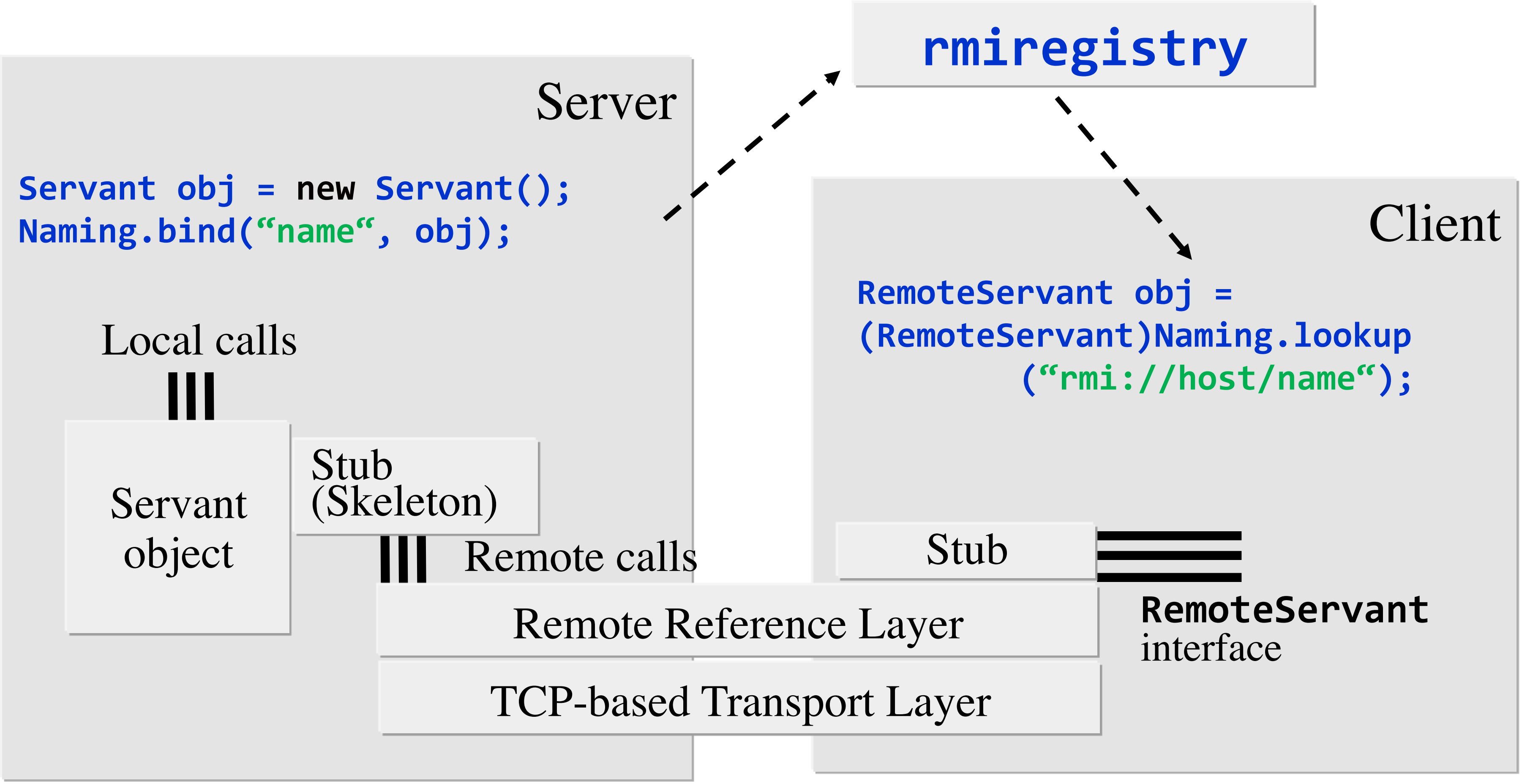
- `java.rmi.Remote`
  - Interface that indicates interfaces whose methods may be invoked from a non-local JVM -- remote interfaces.
- `java.rmi.Naming`
  - The RMI Naming Service client that is used to bind a name to an object and to lookup an object by name.
- `java.rmi.RemoteException`
  - The common superclass for a number of communication-related RMI exceptions.
- `java.rmi.server.UnicastRemoteObject`
  - A class that indicates a non-replicated remote object.

Developing and Executing a Distributed Application with Java RMI

Typical development and execution steps

1. Define a remote interface(s) that extends `java.rmi.Remote`.
2. Develop a class (a.k.a. servant class) that implements the interface.
3. Develop a server class that provides a container for servants, i.e. creates the servants and registers them at the Naming Service.
4. Develop a client class that gets a reference to a remote object(s) and calls its remote methods.
5. Compile all classes and interfaces using `javac`.
6. (Optional) Generate stubs for the servant classes by `rmic`.
7. Start the Naming service `rmiregistry`
8. Start the server on a server host, and run the client on a client host.

Architecture of a Client-Server Application with Java RMI



```
Servant obj = new Servant();
Naming.bind(“name“, obj);
```

```
RemoteServant obj =
(RemoteServant)Naming.lookup
        (“rmi://host/name“);
```

Declaring and Implementing a Remote Interface

- A remote interface must extend `java.rmi.Remote`
  - Each method must throw `java.rmi.RemoteException`
- A class may implement one or several remote interface
  - The class either extends the `UnicastRemoteObject` class or its instance is exported via the static call `UnicastRemoteObject.exportObject(Remote obj)`
- An object of the class that implements the remote interface is called a **servant**.
  - A servant is created by a server and lives until the server dies.
  - The servant and the server can be encapsulated into one class (typically, a primary class).
- A **stub** and a **skeleton** are generated from a servant class automatically

The Naming Service **rmiregistry**. The Naming Client **Naming**

- A Remote object can be registered with a specified name at the Naming service, **rmiregistry**, provided in J2SE.
  - A name can be specified as a URL of the form **rmi://**host:port**/**objectName
  - The URL indicates host/port of rmiregistry (defaults to localhost:1099).
- The **Naming** class provides a static client of the RMI registry.

- A server binds a name to an object:

```
try {
  Bank bank = new BankImpl(“CityBank”);
  Naming.rebind(“rmi://” + host + ”:” + port +
    ”/CityBank”, bank);
  System.out.println(bank + " is ready.");
} catch (Exception e) {
    e.printStackTrace();
}
```

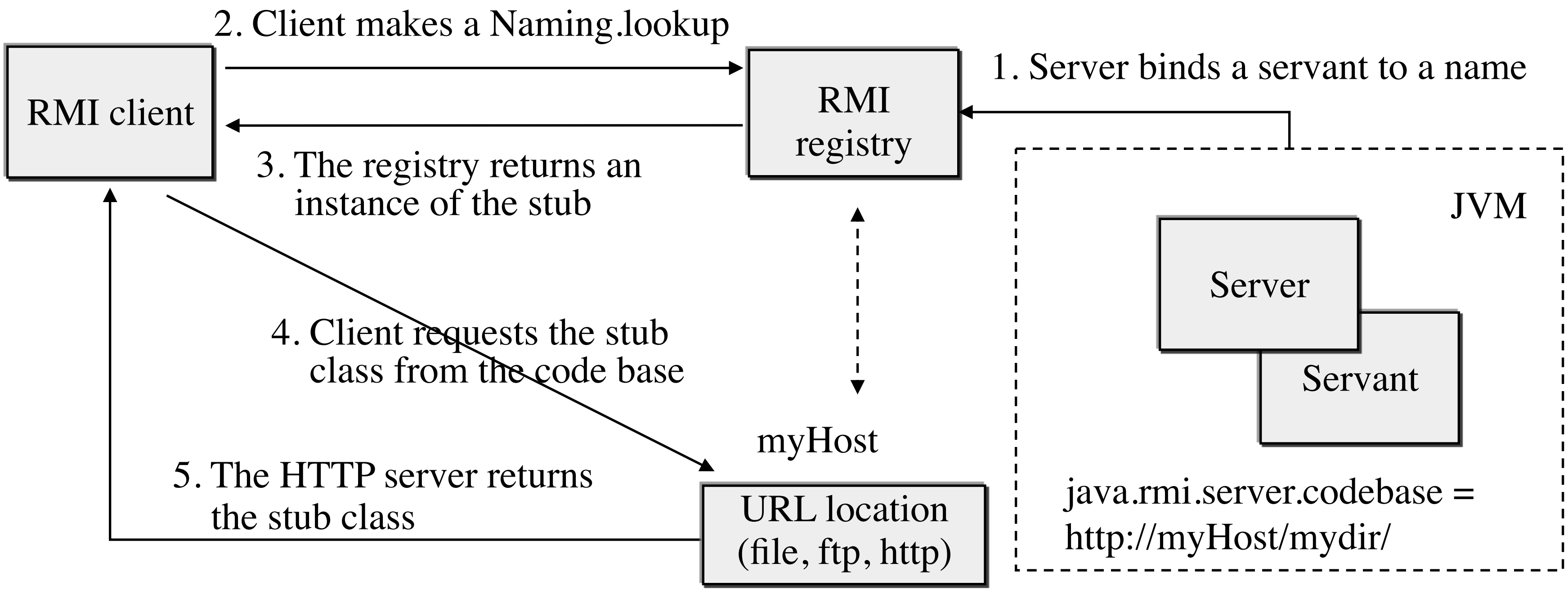
- A client looks up a remote object by name:

```
String rmiregURL = “rmi:// + host;
try {
 bank = (Bank)Naming.lookup(rmiregURL + ”/CityBank”);
} catch (Exception e) {
  System.out.println("The runtime failed: " + e);
  System.exit(0);
}
```

Loading Stub Classes

- **Stubs are dynamically generated and loaded when needed** either from the local file system or from the network using the URL specified on server side using the `java.rmi.server.codebase` property.
  - The property can be set in a command line of an application, for example:

    `-Djava.rmi.server.codebase=http://myHost/mydir/`
  - See: http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-properties.html


2. Client makes a Naming.lookup
RMI client
RMI registry
1. Server binds a servant to a name
3. The registry returns an instance of the stub
JVM
Server
Servant
4. Client requests the stub class from the code base
myHost
5. The HTTP server returns the stub class
URL location (file, ftp, http)
java.rmi.server.codebase = http://myHost/mydir/

Parameters and Returns in Java RMI

- Primitive data types and non-remote **Serializable** objects are passed by values.
  - If an object is passed by value, it is cloned at the receiving JVM, and its copy is no longer consistent with the original object.
  - The class name collision problem. Versioning.
- Remote objects are passed by references.
  - A remote reference can be returned from a remote method. For example:

```
try {
    // lookup for the bank at rmiregistry
    Bank bankobj = (Bank)Naming.lookup(bankname);
    // create a new account in the bank
    Account account = bankobj.newAccount(clientname);
    account.deposit(value);
} catch (Rejected e) { process the exception }
…
```

  - A remote object reference can be passed as a parameter to a remote method.

Example: A Bank Manager

- An application that controls accounts.
- Remote interfaces:
  - `Account` – deposit, withdraw, balance;
  - `Bank` – create a new account, delete an account, get an account;
- Classes that implement the interfaces:
  - `BankImpl` – a bank servant class the implements the `Bank` interface used to create, delete accounts;
  - `AccountImpl` – a account servant class implements the `Account` interface to access accounts.

ROYAL INSTITUTE
OF TECHNOLOGY

Bank and Account Remote Interfaces

- The **Bank** interface

```
package bankrmi;
import java.rmi.*;
import bankrmi.Account;
import bankrmi.Rejected;
public interface Bank extends Remote {
  public Account newAccount(String name) throws RemoteException, Rejected;
  public Account getAccount (String name) throws RemoteException;
  public boolean deleteAccount(String name) throws RemoteException, Rejected;
}
```

- The **Account** interface

```
package bankrmi;
import java.rmi.*;
import bankrmi.Rejected;
public interface Account extends Remote {
  public float balance() throws RemoteException;
  public void deposit(float value) throws RemoteException, Rejected;
  public void withdraw(float value) throws RemoteException, Rejected;
}
```

A Sketch of a Bank Implementation

```
package bankrmi;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
import java.rmi.*;
import bankrmi.*;
public class BankImpl extends UnicastRemoteObject implements Bank {
  private String _bankname = "Noname";
  private Hashtable _accounts = new Hashtable(); // accounts
  public BankImpl(String name) throws RemoteException {
    super(); _bankname = name;
  }
  public BankImpl()  throws RemoteException {
    super();
  }
  public synchronized Account newAccount(String name) throws RemoteException, Rejected {
    AccountImpl account = (AccountImpl) _accounts.get(name);
    if (account != null) {
      System.out.println("Account [" + name + "] exists!!!");
      throw new Rejected("Rejected: Bank: " + bankname +  " Account for: " + name + " already exists: "
    + account);
    }
    account = new AccountImpl(name);
    _accounts.put(name, account);
    System.out.println("Bank: " + _bankname + " Account: " + name + " Created for " + name);
    return (Account)account;
  }
  . . .
```

A Sketch of an Account Implementation

```
package bankrmi;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.*;
import bankrmi.*;
public class AccountImpl extends UnicastRemoteObject implements Account {
  private float _balance = 0;
  private String _name = "noname";
  public AccountImpl(String name) throws RemoteException {
    super();
    this.name = name;
  }
  public AccountImpl() throws RemoteException  {
    super();
  }
  public synchronized void deposit(float value) throws RemoteException, Rejected {
    if (value < 0) throw new Rejected("Rejected: Account " + name + ": Illegal value: " + value);
    _balance += value;
    System.out.println("Transaction: Account "+name+": deposit: $" + value + ", balance: $" + _balance);
  }
  public synchronized void withdraw(float value) throws RemoteException, Rejected {
   …
  }
  public synchronized float balance() throws RemoteException { return _balance; }
}
```

The Server Application

```
package bankrmi;
import java.rmi.*;
import bankrmi.*;
public class Server {
  static final String USAGE = "java bankrmi.Server <bank_url>";
  static final String BANK = "NordBanken";
  public Server(String[] args) {
    String bankname = (args.length > 0)? args[0] : BANK;
    if (args.length > 1 || bankname.equalsIgnoreCase("-h")) {
      System.out.println(USAGE);
      System.exit(1);
    }
    try {
      Bank bankobj = (Bank)(new BankImpl(bankname));
      Naming.rebind(bankname, bankobj);
      System.out.println(bankobj + " is ready.");
    } catch (Exception e) { System.out.println(e); }
    Object sync = new Object();
    synchronized(sync) { try { sync.wait();} catch (Exception ie) {}}
  }
  public static void main(String[] args) {
    new Server(args).start();
  }
}
```

A Sketch of a Client Application

```
package bankrmi;
import bankrmi.*;
import java.rmi.*;
public class SClient {
    static final String USAGE = "java Client <bank_url> <client> <value>";
    String bankname = “Noname", clientname = “Noname"; // defaults
    float value = 100;
    public SClient(String[] args) {
        // Read and parse command line arguments (see Usage above)
        ...
        try {
          Bank bankobj = (Bank)Naming.lookup(bankname);
          Account account = bankobj.newAccount(clientname);
          account.deposit(value);
          System.out.println (clientname + "'s account: $" + account.balance());
        } catch (Exception e) {
          System.out.println("The runtime failed: " + e);
          System.exit(0);
        }
    }
    public static void main(String[] args) {
      new SClient(args);
    }
}
```