

# **Konzepte objektorientierter Programmierung – Lecture 9 –**

**Prof. Dr. Peter Müller**  
Software Component Technology

Wintersemester 04/05

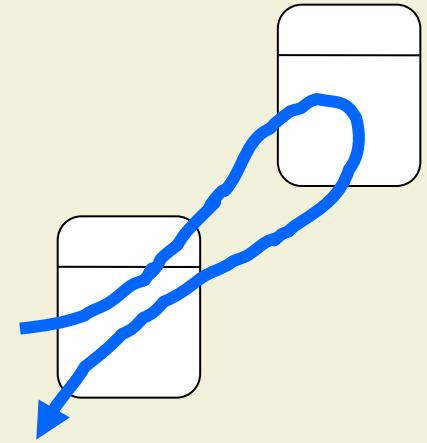
**ETH**

Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Concurrency in OO-Programs

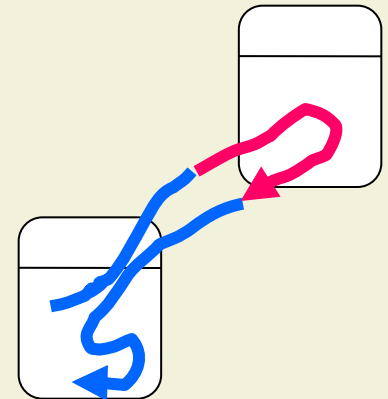
## ■ Passive objects

- Threads **pass through different objects** (by method invocations)
- **Several threads** executed **on one object** possible



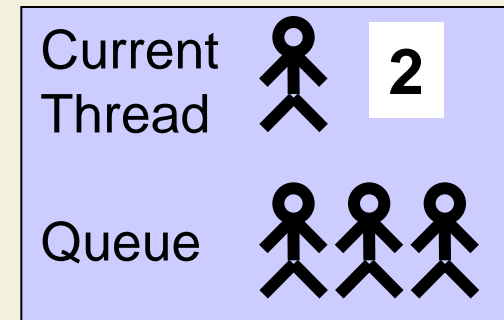
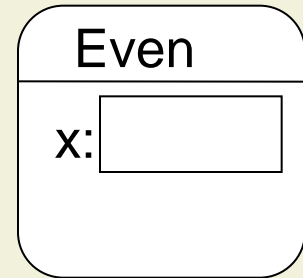
## ■ Active objects

- **Each object has** its own **thread**
- Upon method invocation, the thread of the target object serves the request
- **At most one thread** executed on one object



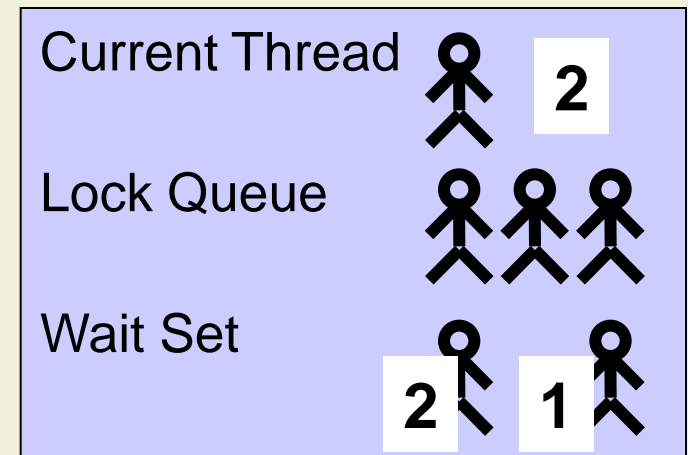
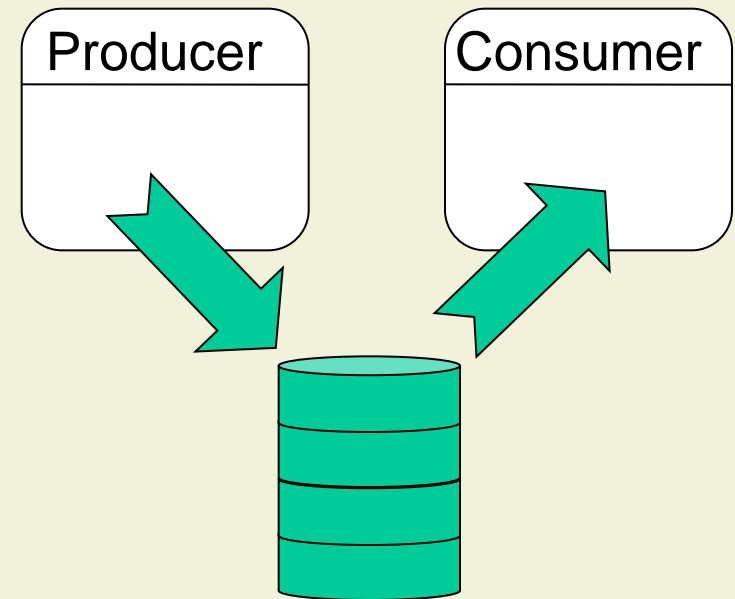
# Object-Oriented Monitors

- Each object has a monitor
- Execution of synchronized methods requires lock of monitor
  - Lock is obtained upon invocation
  - Lock is released upon termination
  - Other threads have to wait
- Monitor keeps track of
  - Thread that has locked the monitor
  - Number of locks of this thread
  - Queue of blocked threads



# Wait and Notify

- Wait operation
  - Can be applied if a thread has locked a monitor
  - Puts thread into wait state and adds thread to wait set
  - Releases lock
- Notify operation
  - Can be applied if a thread has locked a monitor
  - Chooses one thread from wait set and re-enables it for scheduling



# Agenda for Today

## 9. Distribution

9.1 Sockets

9.2 Serialization

9.3 Remote Objects

9.4 Middleware Architectures

9.5 Heterogeneous Environments

## Objectives

- Remote objects
- Remote method invocation

# Aspects of Distributed Programming

- Programs run in **different processes** or on different computers
  - Usually no shared memory
- **Communication** is crucial
  - Communication is not robust
  - Communication takes time
- Distributed systems are often **heterogeneous**
  - Different hardware
  - Different operating systems
  - Different programming languages

# 9. Distribution

## 9.1 Sockets

## 9.2 Serialization

## 9.3 Remote Objects

## 9.4 Middleware Architectures

## 9.5 Heterogeneous Environments

# Producer-Consumer Example

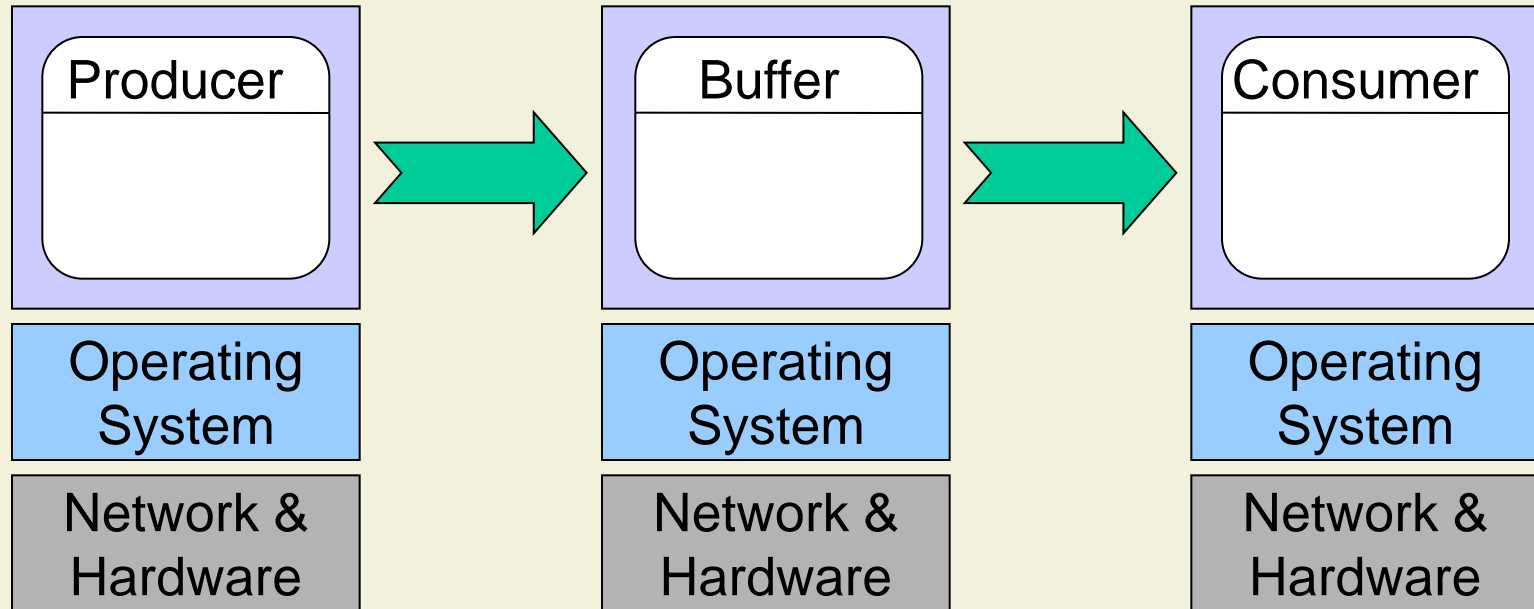
```
class Buffer {  
    ...  
    synchronized void put( Prd p ) {  
        if ( isFull( ) )    wait( );  
        ...  
        notify( );  
    }  
  
    synchronized Prd get ( ) {  
        if ( isEmpty( ) ) wait( );  
        ...  
        notify( );  
    }  
}
```

```
class Producer extends Thread {  
    Buffer buf;  
    void run( ) {  
        while ( true )  
            buf.put( new Prd( ) );  
    }  
}
```

```
class Consumer extends Thread {  
    Buffer buf;  
    void run( ) {  
        while ( true )  
            buf.get( );  
    }  
}
```

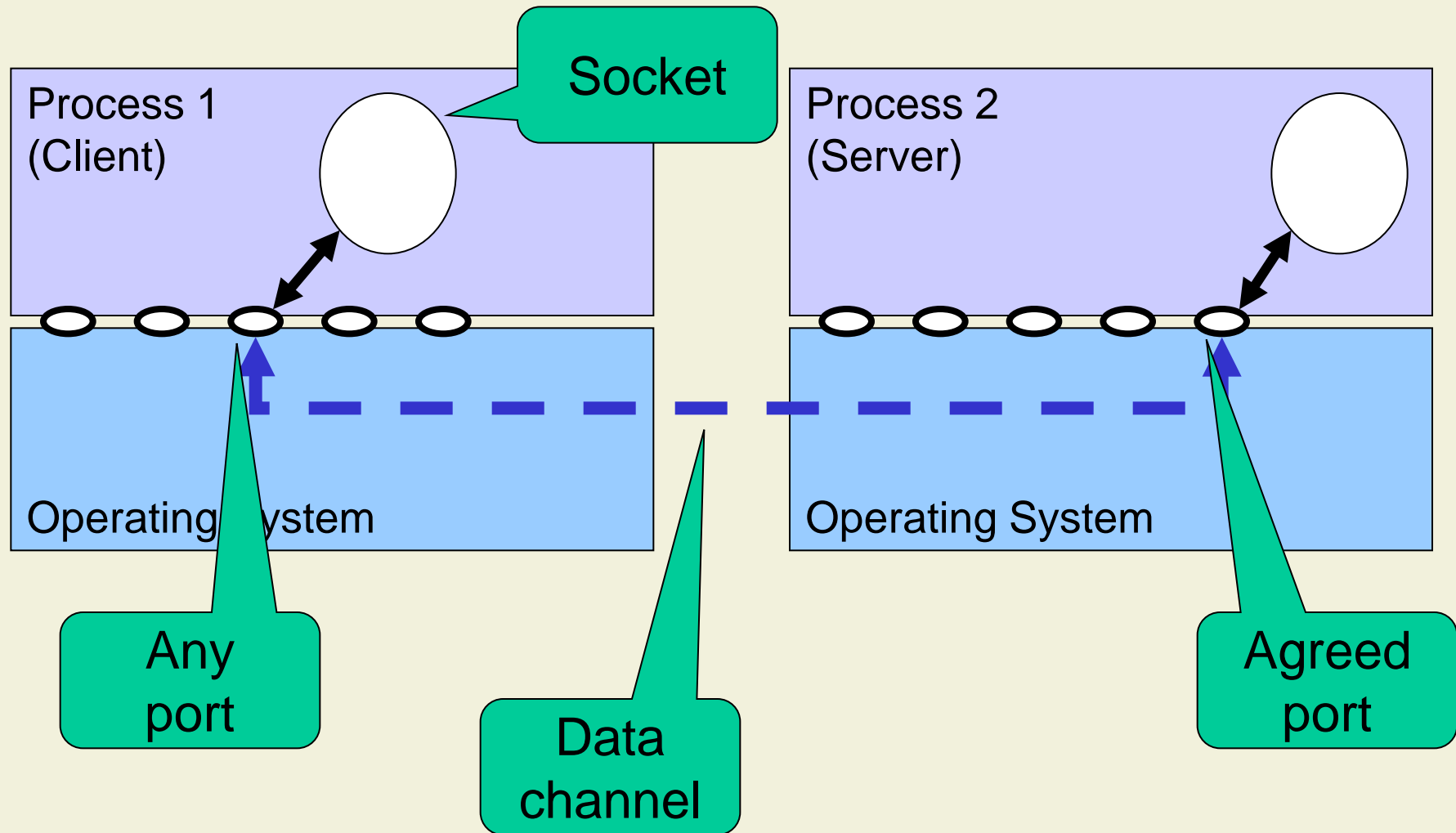


# Distributed Producer-Consumer Example

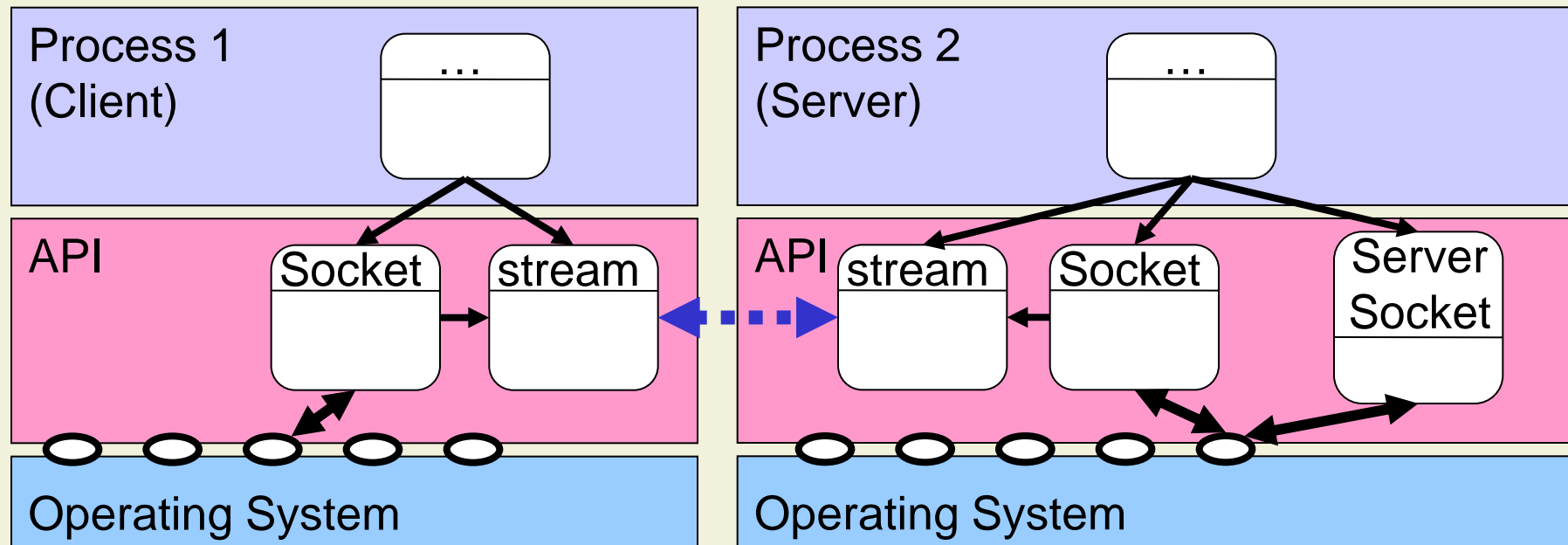


- How to access objects in different address spaces?
- How to communicate across process boundaries?
- How to pass parameters, results, and exceptions?

# Sockets and Ports



# Communication via Sockets



- Server sockets wait for communication partners on an agreed port
- Sockets provide communication facilities
- Input and output streams to transmit data

# Example: Buffer with Socket

```
class Buffer {  
    ...  
    synchronized void put( String p ) { ... }  
    synchronized String get ( )      { ... }  
}
```

```
class BufferServer {  
    static Buffer buf = new Buffer( );  
    static void main( ... ) {  
        ServerSocket ss = new ServerSocket( 1199 );  
        while ( true ) {  
            Socket s = ss.accept( );  
            new ServiceThread( s ).start( );  
        } } }
```

- Buffer remains unchanged
- Main loop accepts clients and starts new threads
- Code does not show exception handling

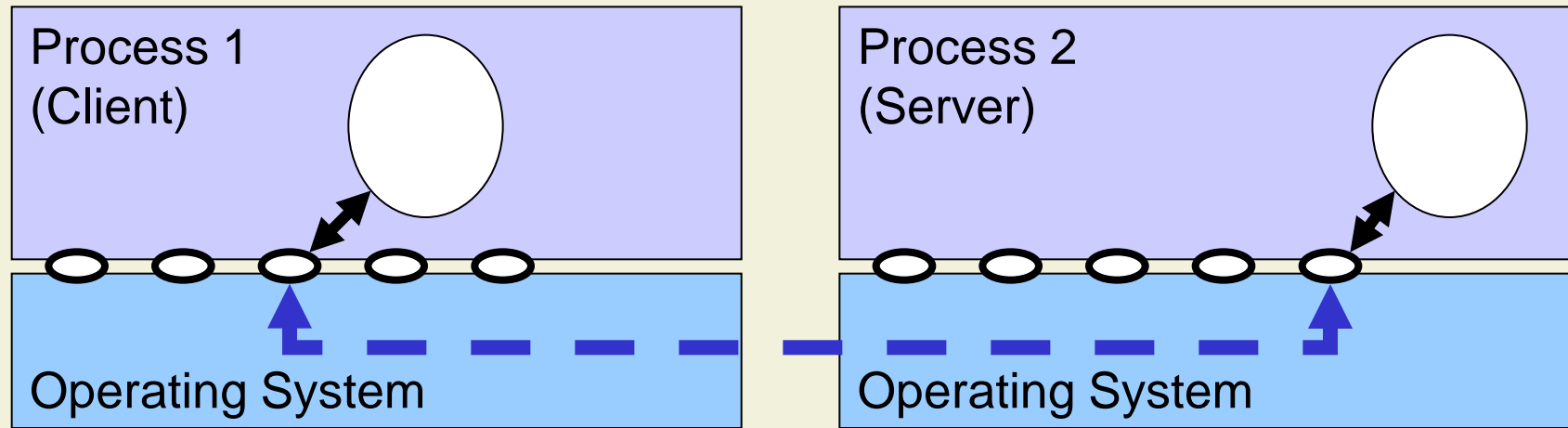
# Example: Buffer with Socket (cont'd)

```
class ServiceThread extends Thread {  
    Socket s;  
    ServiceThread( Socket p ) { s = p; }  
  
    void run( ) {  
        BufferedReader br =  
            new BufferedReader( new InputStreamReader( s.getInputStream( ) )  
        );  
        if ( br.readLine( ).equals( "put" ) )  
            BufferServer.buf.put( br.readLine( ) );  
        else {  
            PrintWriter pw = new PrintWriter( s.getOutputStream( ),true );  
            pw.println( BufferServer.buf.get( ) );  
        }  
    }  
}
```

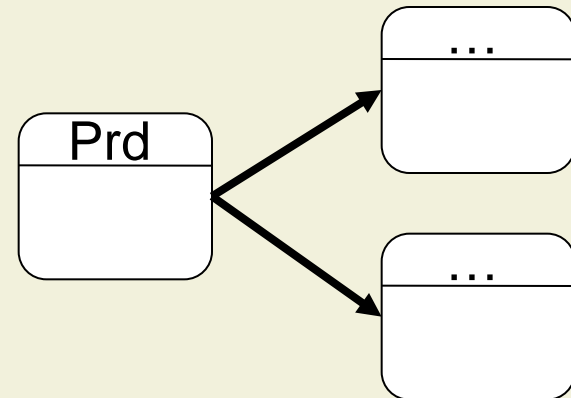
# Example: Producer with Socket

```
class Producer extends Thread {  
    String computer; int port;  
    Producer( String c, int p ) { computer = c; port = p; }  
    void run( ) {  
        while ( true ) {  
            Socket s = new Socket( computer, port );  
            PrintWriter pw = new PrintWriter( s.getOutputStream( ),true );  
            pw.println( "put" );  
            pw.println( "Product" );  
            sleep( 1000 );  
        } }  
    static void main( ... ) { new Producer( "monkey",1199 ).start( ); }  
}
```

# Parameter Passing



- Commands, parameters, results, and exceptions are transmitted as sequential byte streams



# 9. Distribution

9.1 Sockets

**9.2 Serialization**

9.3 Remote Objects

9.4 Middleware Architectures

9.5 Heterogeneous Environments



# Serialization and Deserialization

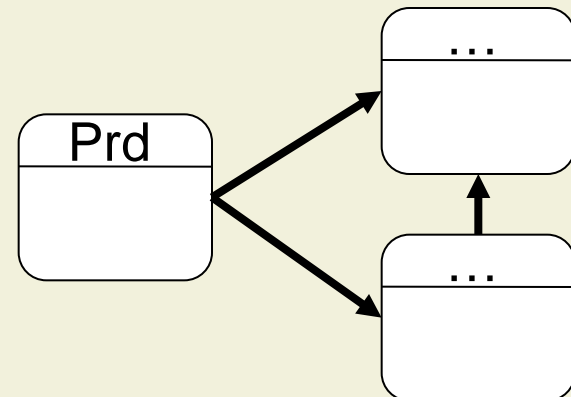
- Serialization transforms object structures into a **sequential format**
- Sequential format is **independent of memory addresses**
- Serialization is used
  - To save object structures persistently
  - To exchange object structures between address spaces
- Often called marshalling and unmarshalling

# Object Streams in Java

- Serialization needs access to private fields
  - Interface Serializable is used as tag
- Object streams serialize
  - Values of primitive types
  - Serializable objects
- All objects except strings are written only once

```
interface Serializable { }
```

```
class ObjectOutputStream  
    extends OutputStream  
    implements ... {  
  
    void writeObject( Object obj )  
        throws IOException { ... }  
    ... }
```



# Producer with ObjectOutputStream

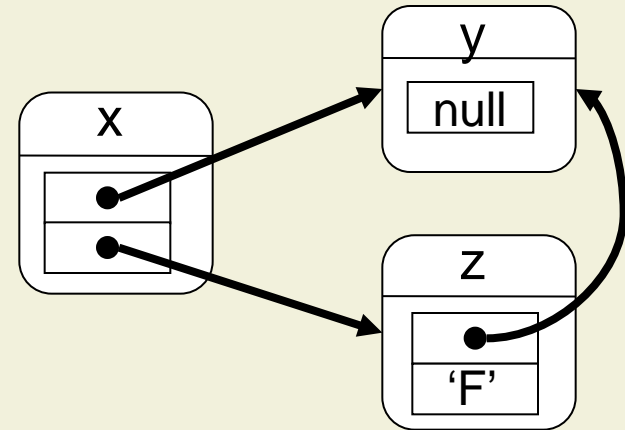
```
class Prd implements Serializable { ... }
```

```
class Producer extends Thread {  
    ...  
    void run( ) {  
        while ( true ) {  
            Socket s = new Socket( computer, port );  
            ObjectOutputStream out =  
                new ObjectOutputStream( s.getOutputStream( ) );  
            out.writeObject( "put" );  
            out.writeObject( new Prd( ) );  
            sleep( 1000 );  
        }  
    }  
}
```

# How Object Streams Work (Simplified)

## Simplified Algorithm

```
proc serialize( o )  
  if o is already serialized then  
    lookup and write id of o  
  else  
    assign unique id to o  
    enter o and id to table  
    write id of o  
    write fields of o
```

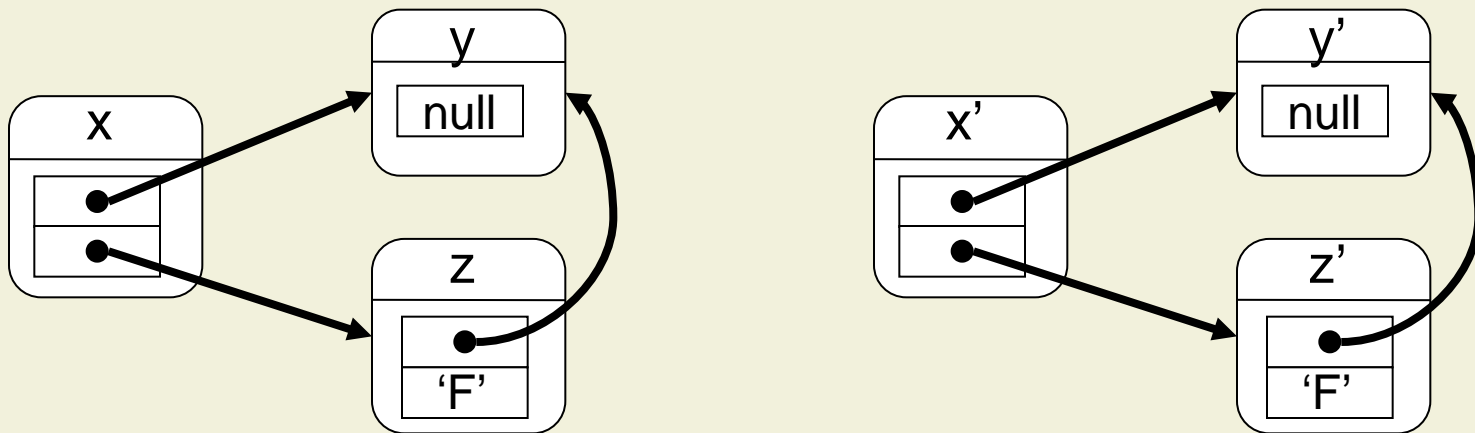


Object	Id
x	1
y	2
z	3

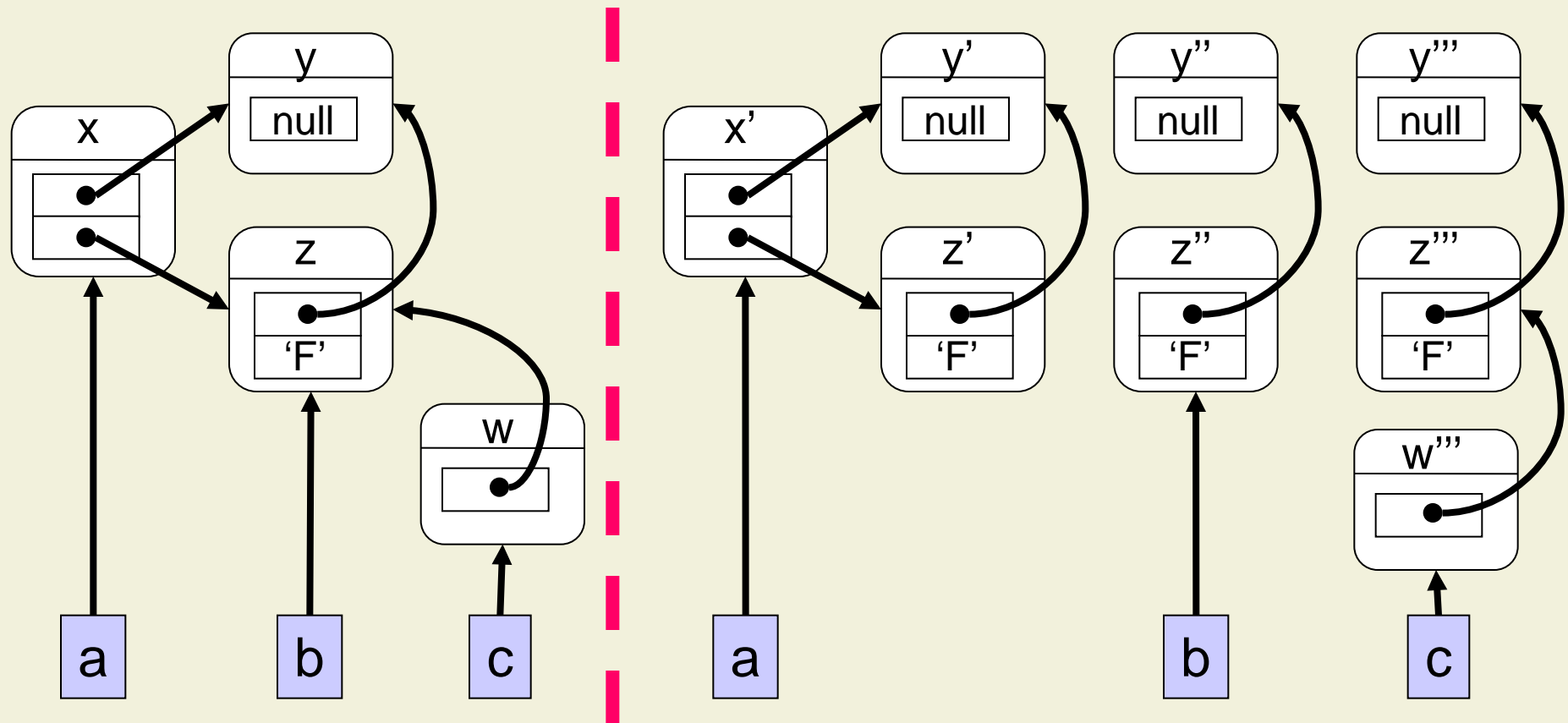
Stream: 1 – 2 – null – 3 – 2 – F

# Object Identity

- Serialization and deserialization
  - **Preserve “relative” object identities** within object structures (except strings)
  - **Do** (of course) **not preserve absolute object identities**
- Consequences for **side-effects** and comparison



# Aliasing



- Only reachable objects are serialized
- Serialization can destroy aliasing properties

# Discussion of Socket Solution

- Communication has to be coded explicitly
  - Commands, parameters, results, exceptions
- No static type safety
- Loss of object identities
- Significantly different from local solution

```
while ( true )  
  buf.put( new Prd( ) );
```

```
while ( true ) {  
  Socket s = new Socket( computer, port );  
  ObjectOutputStream out = new  
    ObjectOutputStream( s.getOutputStream( ) );  
  out.writeObject( "put" );  
  out.writeObject( new Prd( ) );  
}
```

# 9. Distribution

9.1 Sockets

9.2 Serialization

**9.3 Remote Objects**

9.4 Middleware Architectures

9.5 Heterogeneous Environments



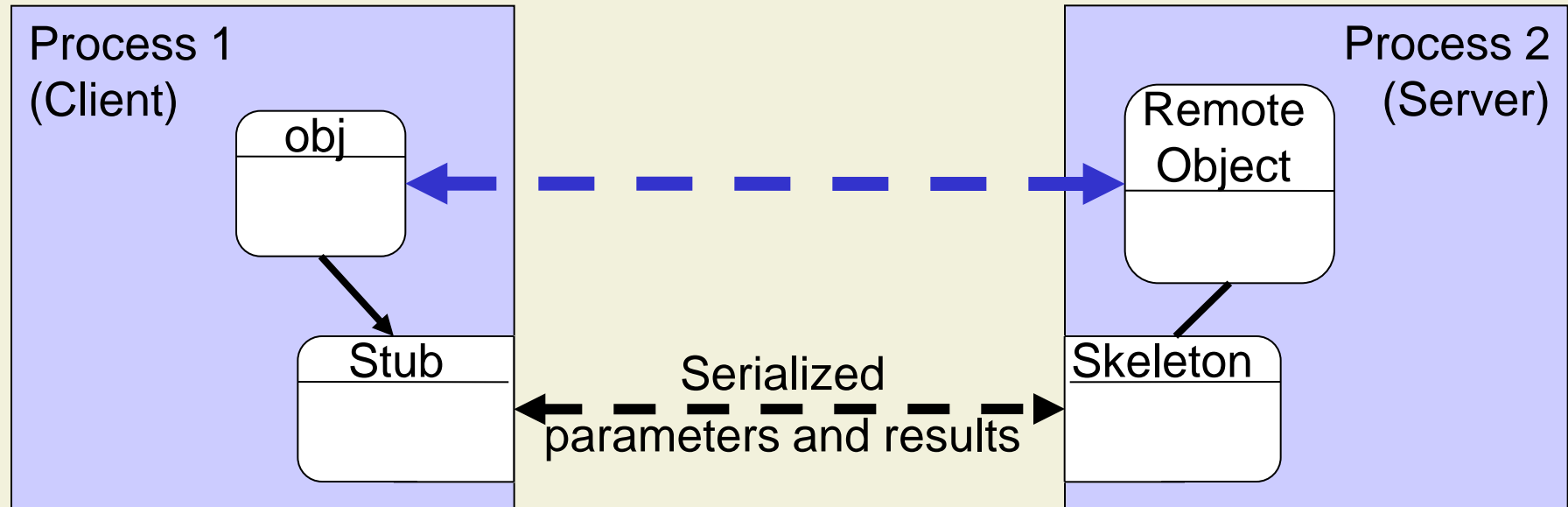
# The Holy Grail Of Remote Computing

[From Charlie Sawyer's course "*Java For Distributed Programming*", Harvard University, 2003]



No, Sire, it is but a local stub.

# Stubs and Skeletons



- Remote objects are represented locally by stubs
- Stubs and skeletons provide communication
- Code for stubs and skeletons can be generated automatically (RMI compiler `rmic`)

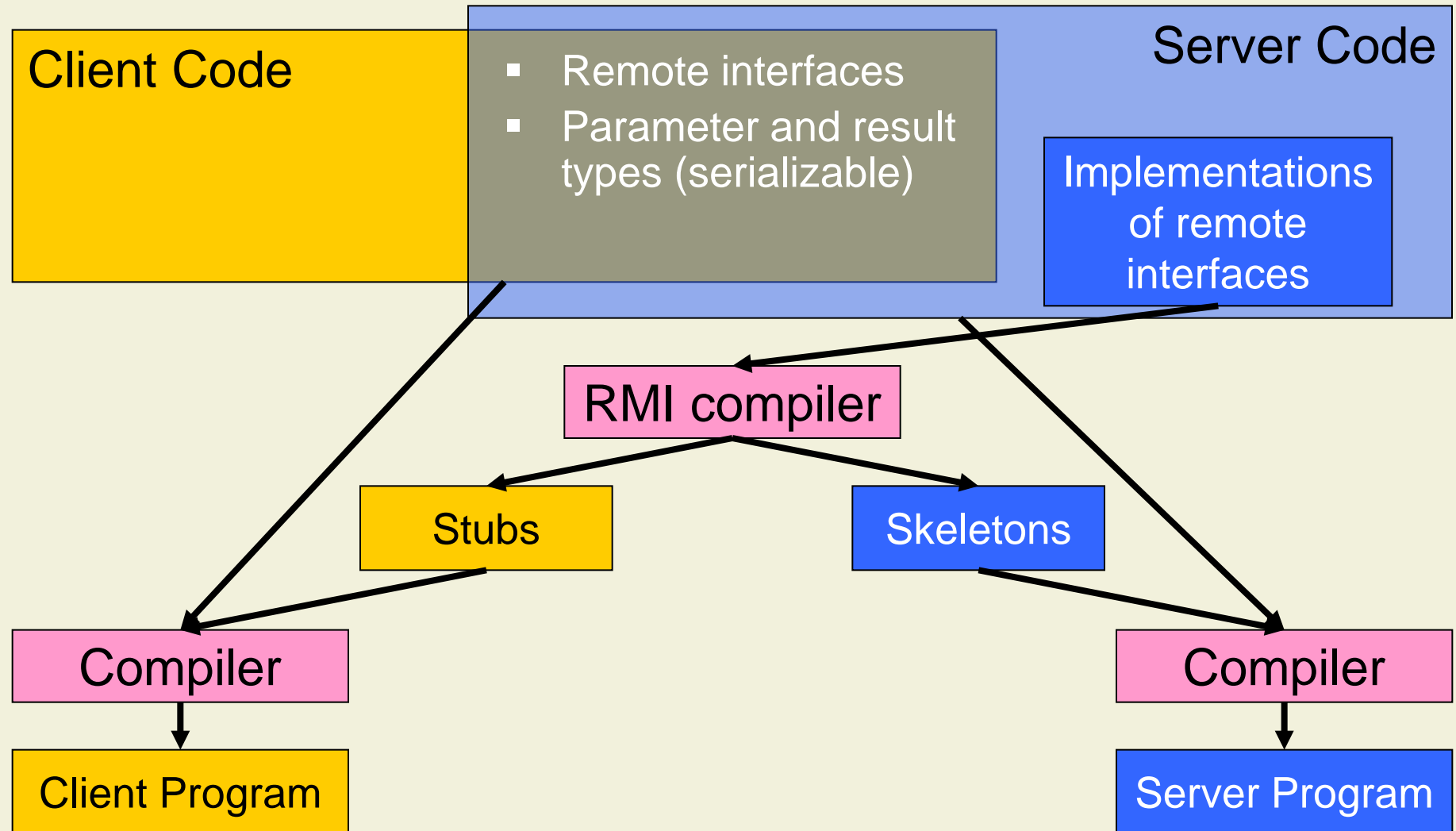
# Remote Interfaces

- Methods that are available remotely must be specified in an interface that extends Remote

```
interface Remote { }
```

```
interface Buffer extends Remote {  
    void put( Prd p )  
        throws RemoteException;  
    Prd get( )  
        throws RemoteException;  
}
```

# Programming with Remote Objects



# Example: Remote Buffer Implementation

```
class BufferImpl
  extends UnicastRemoteObject
  implements Buffer {

  // fields identical to local solution;

  BufferImpl() throws RemoteException { }

  synchronized void put( Prd p )
  { // identical to local solution }
  synchronized Prd get( )
  { // identical to local solution }
}
```

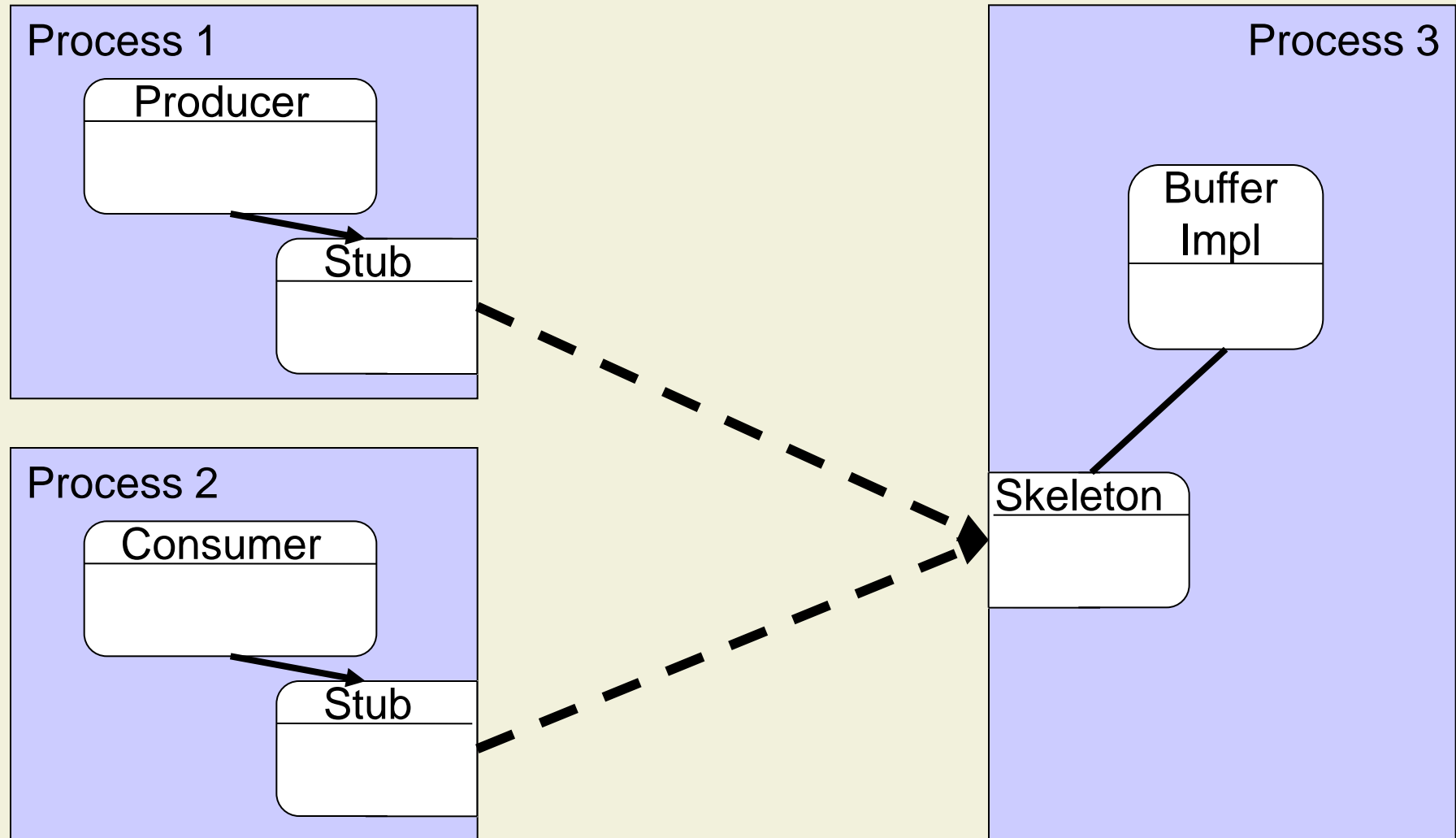
- Implementations of remote objects extend UnicastRemoteObject (or similar classes)
- Constructors may throw exception
- Almost identical to class Buffer of local solutions

# Remote Method Invocation

- Remote interfaces can be used to invoke methods of remote objects
- Communication is transparent except for
  - Error handling
  - Problems of serialization
- Coding is almost identical to local solutions

```
class Producer extends Thread {  
    Buffer buf;  
  
    Producer( Buffer b ) { buf = b; }  
  
    void run( ) {  
        while ( true )  
            try {  
                buf.put( new Prd( ) );  
            } catch( Exception e ) { ... }  
        }  
    }
```

# Process Interaction



# Finding Objects

- References to remote objects are obtained through a **name service**
- **Name server** (rmiregistry) must run on server site
  - Offers service at a certain port
  - Communication with name server is handled by API

```
class Naming {  
    static Remote lookup( String name )    throws ... { ... }  
    static void rebind( String name, Remote obj ) throws ... { ... }  
    ...  
}
```



# Example: BufferServer

```
class BufferServer {  
    static void main( ... ) throws Exception {  
        Naming.rebind( "buffer", new BufferImpl( ) );  
    }  
}
```

```
class Producer extends Thread {  
    ...  
    static void main( ... ) throws Exception {  
        String url = "rmi://monkey/buffer";  
        Buffer b = ( Buffer ) Naming.lookup( url );  
        new Producer( b ).start( );  
    }  
}
```

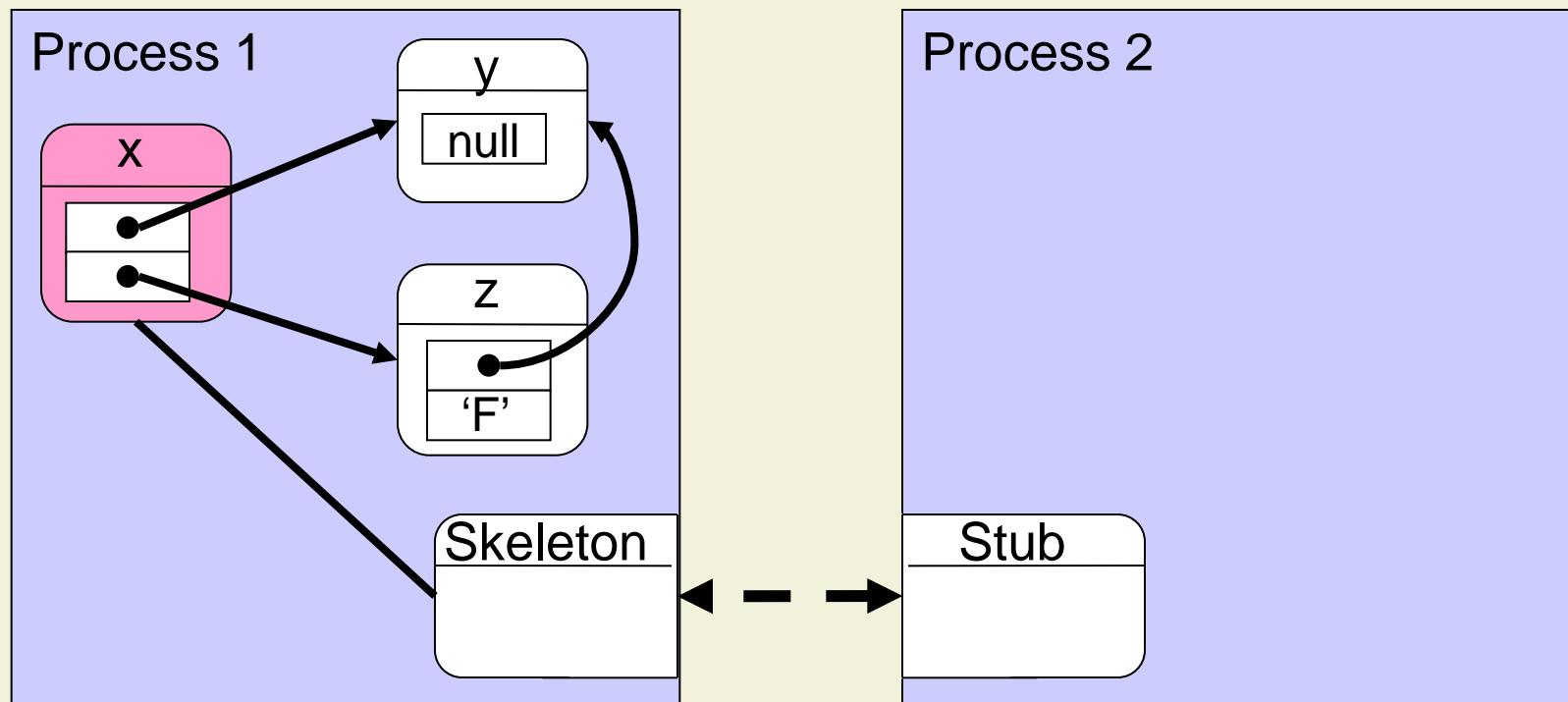
- Remote objects are **registered** at name server
- **URLs** are used to retrieve remote references

# Using RMI in Java

1. Define **interface of remote object** (extends Remote)
2. Define **implementation of remote object** (extends UnicastRemoteObject)
3. Generate **stub and skeleton** classes (rmic)
4. Start **name server** (rmiregistry)
5. Server program **registers remote objects** at registry
6. Client programs **retrieve remote references** through URL (name of computer and name of remote object)

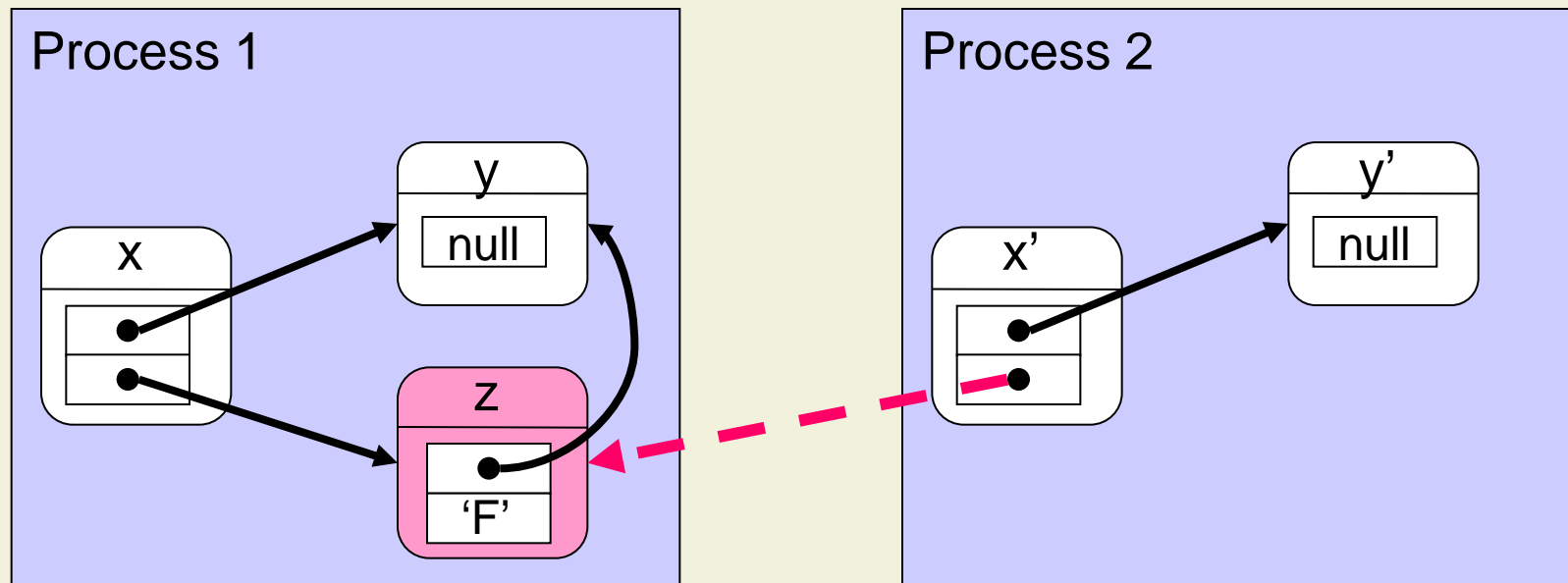
# Serialization of Remote Objects

- Remote objects are not serialized when passed as parameters or results
- Passing remote objects lead to remote references



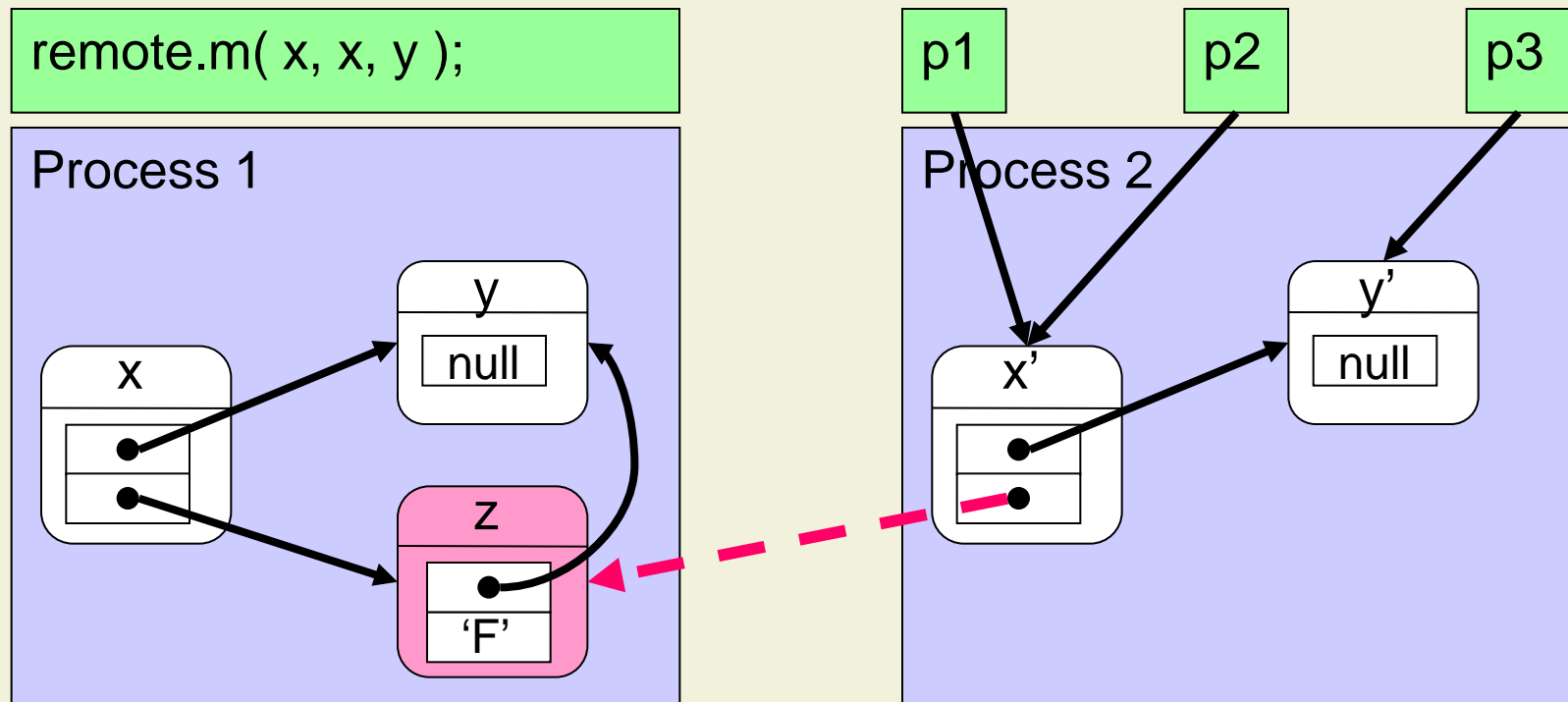
# Details of Serialization

- Remote objects are not serialized when passed as parameters or results
- Rule also applies to remote objects that are referenced indirectly



# Details of Serialization: Aliasing

- Parameters of *one* remote method invocation are serialized together
- Aliases do not lead to duplicate objects



# Remote Objects: Summary

- Remote objects can be accessed similarly to local objects
- Remote objects are accessed through Remote interfaces
  - No field access
  - Only public methods
- Communication is transparent except for
  - Error handling
  - Problems of serialization

# 9. Distribution

9.1 Sockets

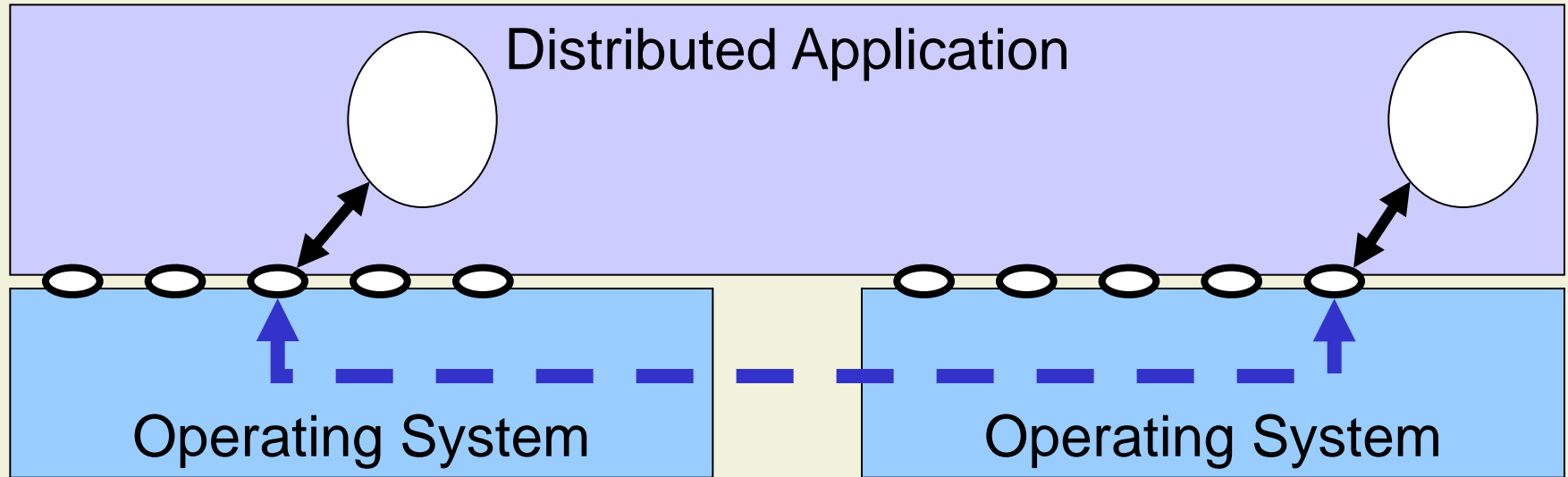
9.2 Serialization

9.3 Remote Objects

**9.4 Middleware Architectures**

9.5 Heterogeneous Environments

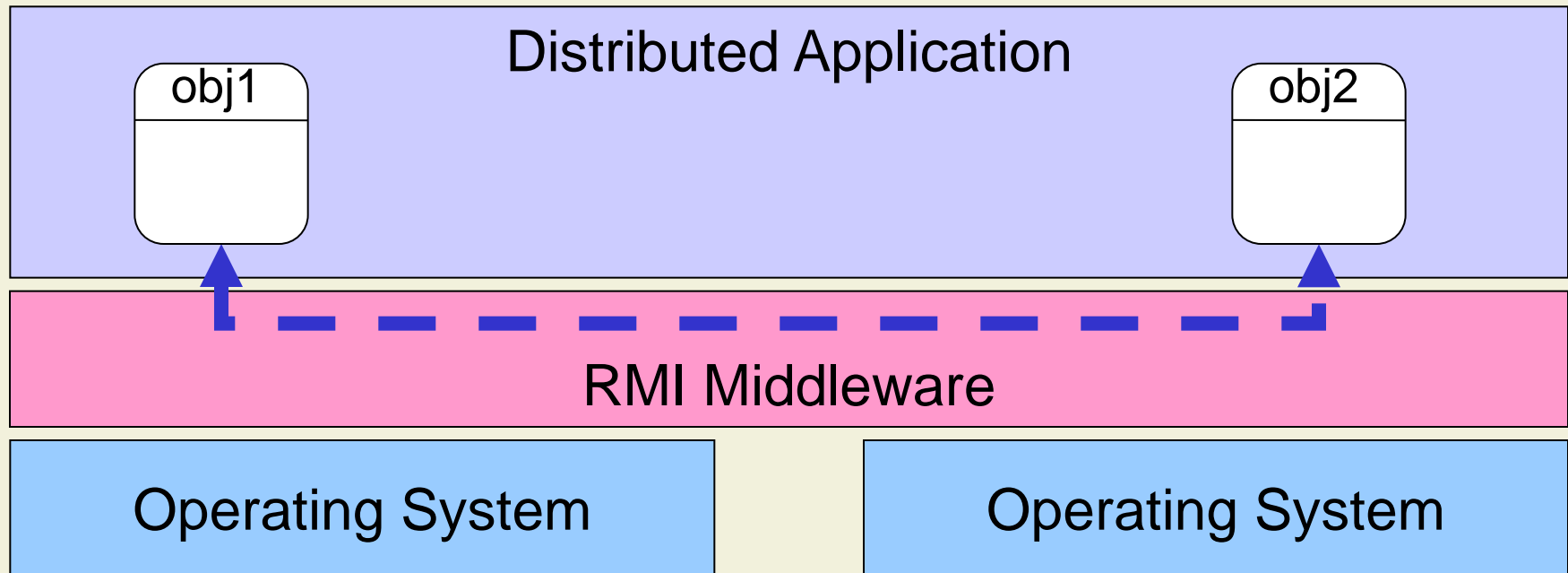
# Socket Communication



- Application has to deal with communication explicitly
  - Locations of objects
  - Communication protocol

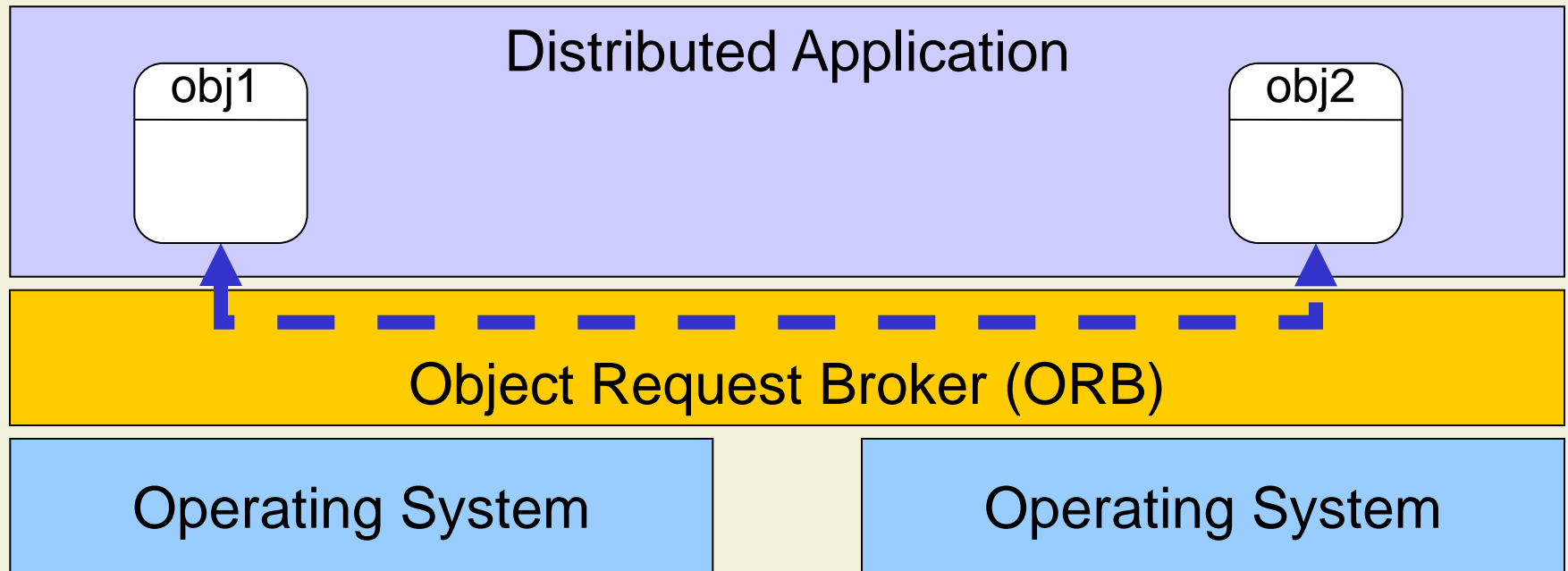


# RMI Middleware



- RMI middleware provides
  - Higher-level abstractions (RMI)
  - Location transparency (almost)
  - Independence of communication protocols

# Request Broker Architectures



- Object request brokers provide
  - Object brokerage
  - Additional higher-level abstractions (multicast, asynchronous communication)

# 9. Distribution

9.1 Sockets

9.2 Serialization

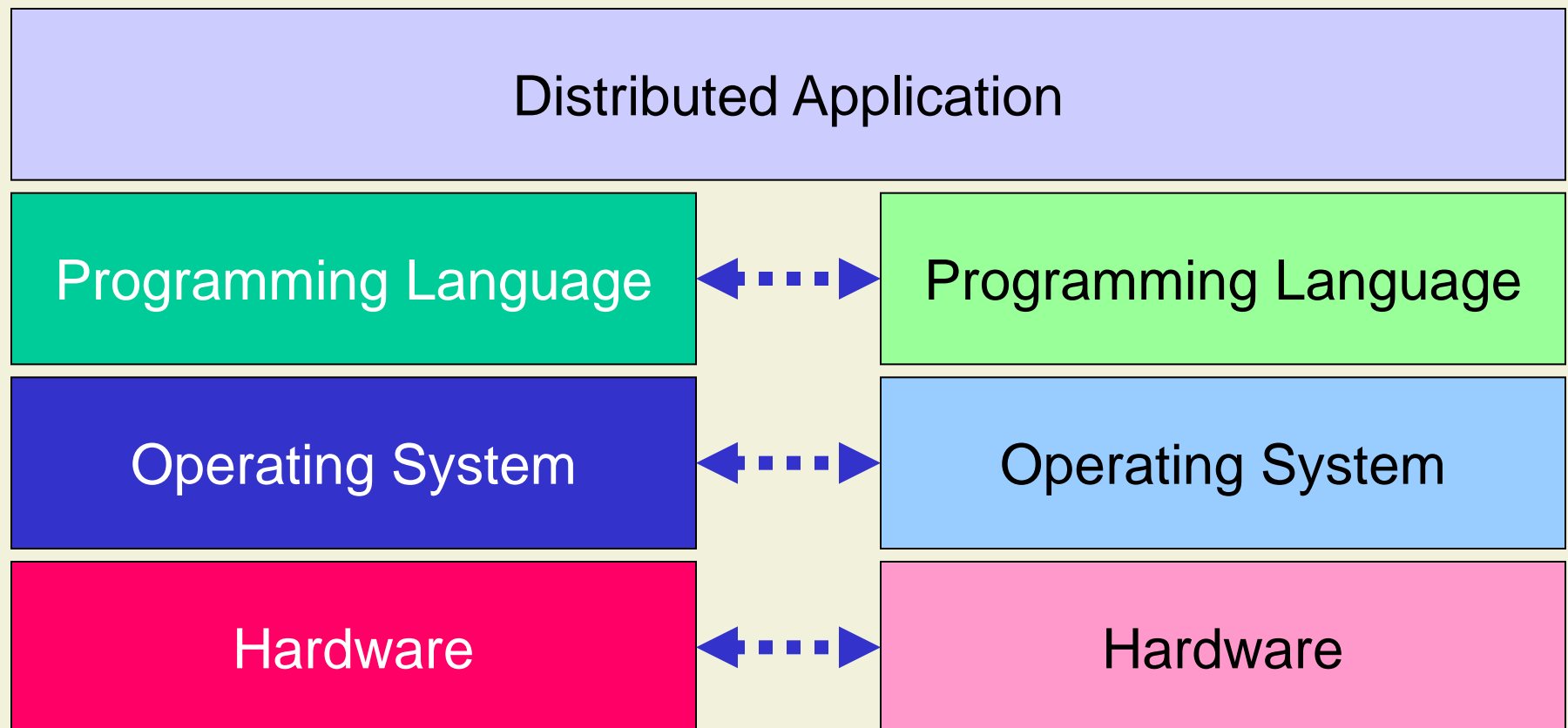
9.3 Remote Objects

9.4 Middleware Architectures

**9.5 Heterogeneous Environments**

# Heterogeneous Environments

- Distributed systems usually run in heterogeneous environments

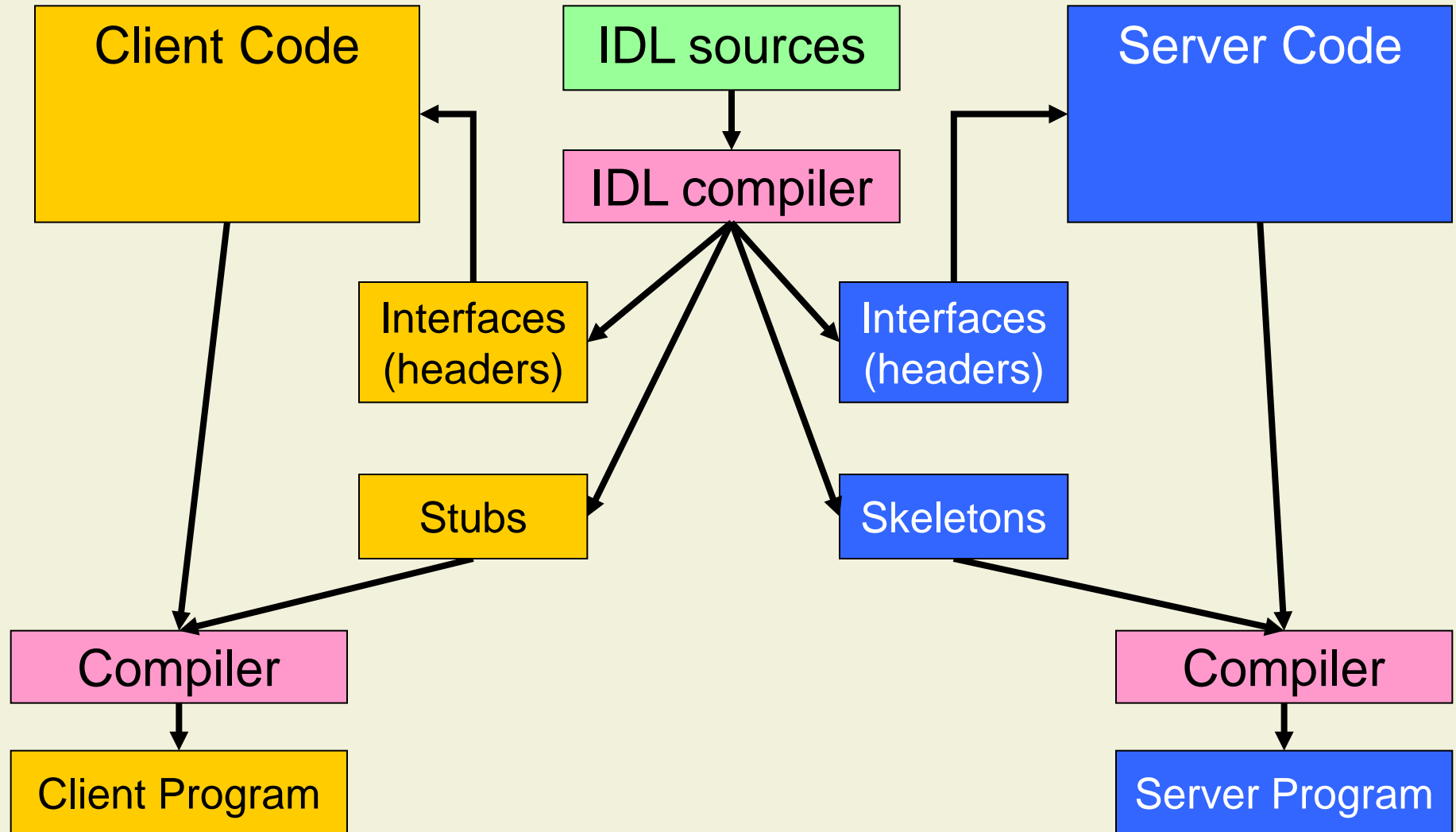


# Interface Description Languages (IDLs)

- Remote objects can be implemented in different languages
- Interfaces have to be described independently from language
- IDL descriptions can be mapped to different programming languages

```
module examples
{
  interface Tribble
  {
    void pet (in long times);
    string make_noise ();
    Tribble reproduce ();
  };
};
```

# Programming with Interface Descriptions



# Serialization of Remote Objects

- Remote objects are not serialized when passed as parameters or results
- Passing remote objects lead to remote references

