

Technology Session II

Serialization, Sockets, RMI

Chair of Programming Methodology

Material based on Prof. Peter Müller's
Konzepte objektorientierter Programmierung
course

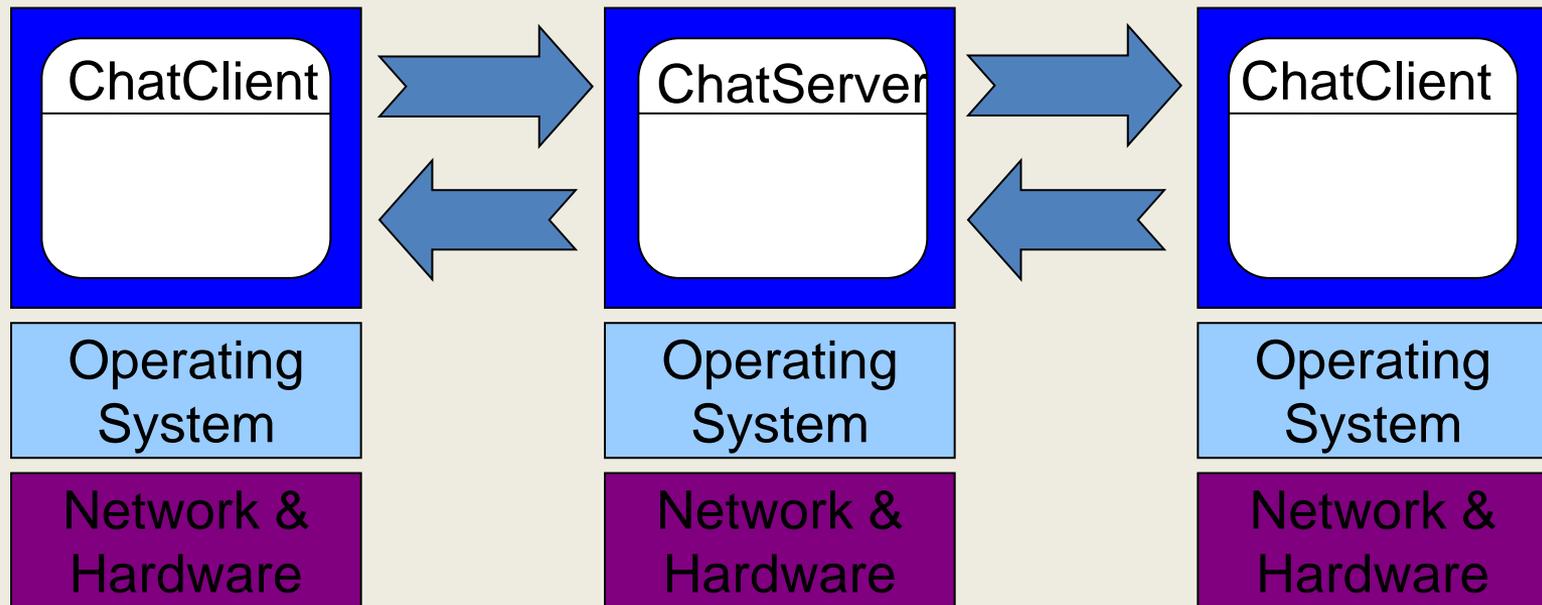
Agenda for Today

- Distributed Programming
 - Sockets
 - Serialization
 - Remote Objects
- 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

Distributed Chat Example



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

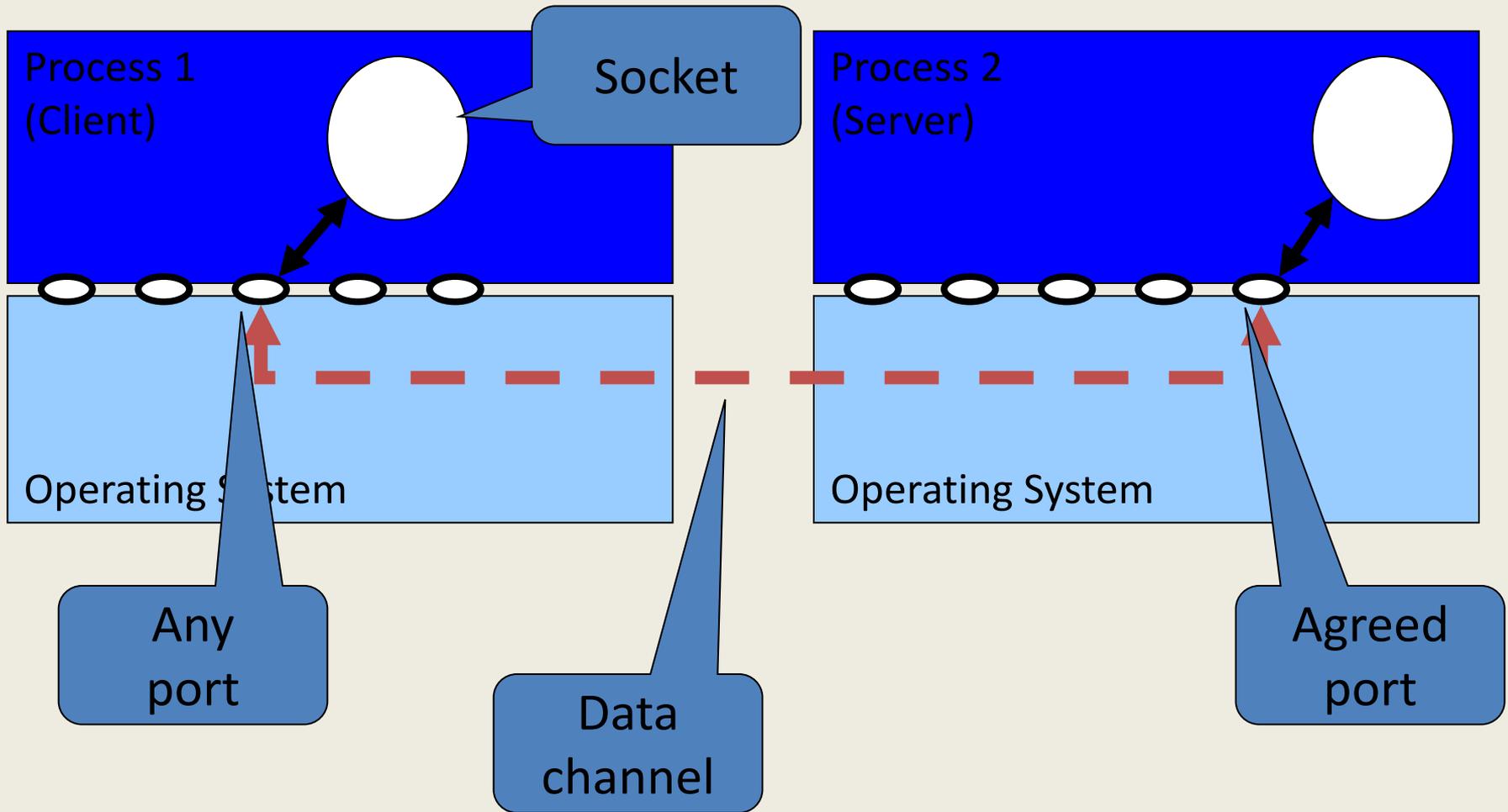
Sockets

Serialization

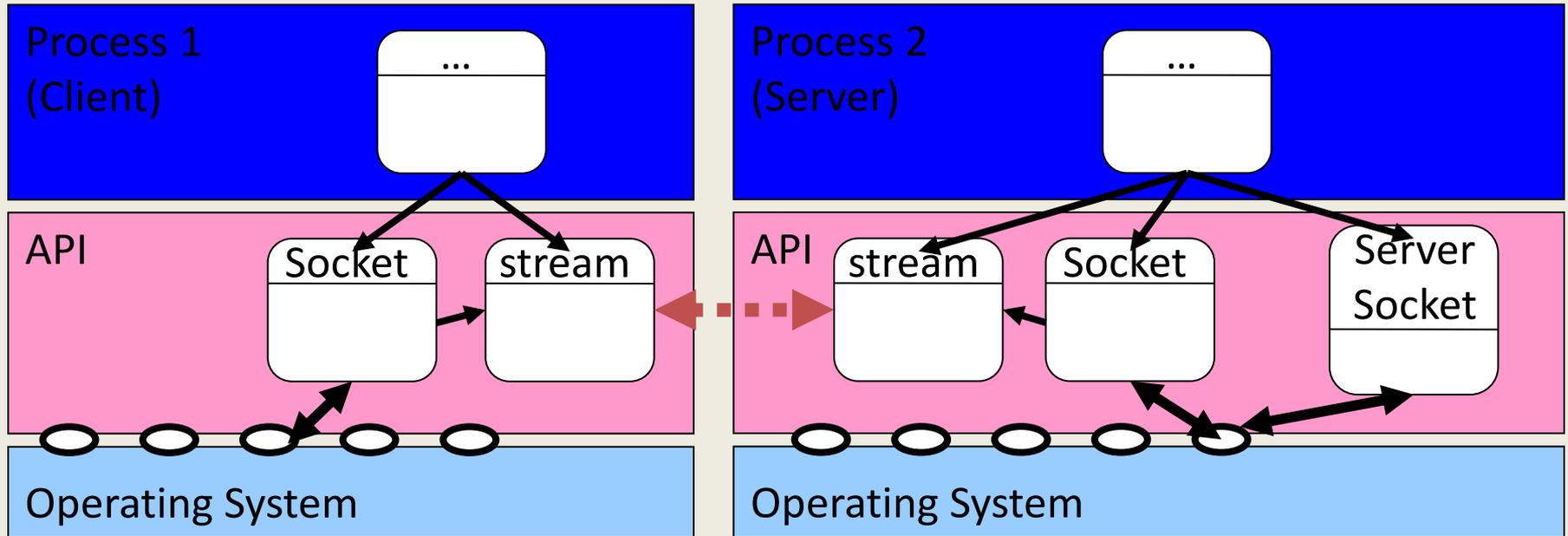
Remote Objects

DISTRIBUTED PROGRAMMING

Sockets and Ports

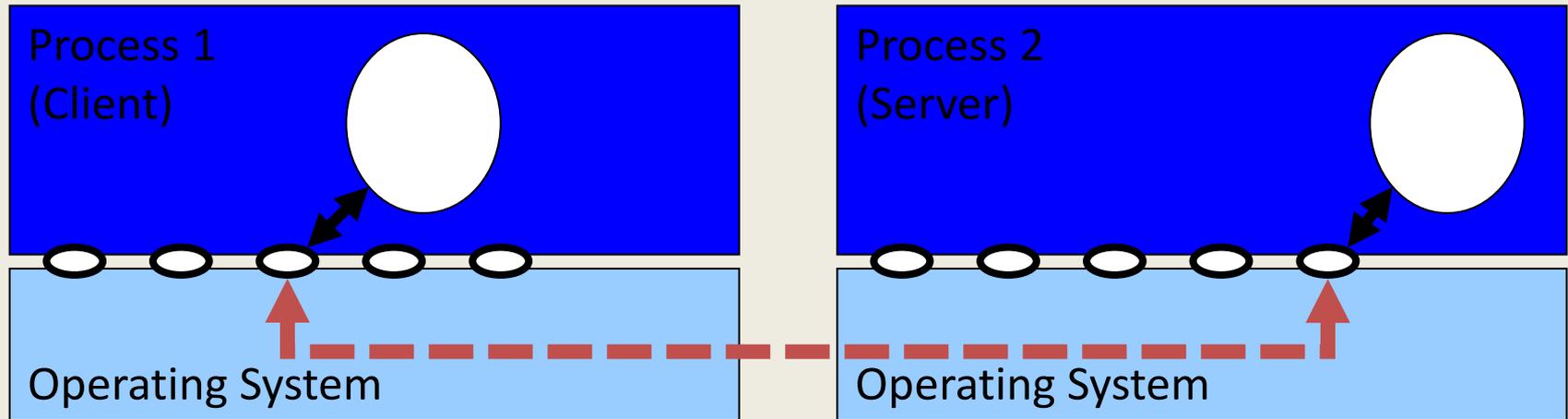


Communication via Sockets



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

Parameter Passing



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

Example: Chat server with Socket

```
public class SocketChatServer {  
  
    public static void main( String[] args ) {  
        SocketChatServer srv = new SocketChatServer();  
        ServerSocket ss = new ServerSocket( 6666 );  
        while ( true ) {  
            Socket s = ss.accept( );  
            new ServiceThread( s ).start();  
        }  
    }  
}
```

- Main loop accepts clients and starts new threads.
- Code does not show exception handling.

Example: Chat server with Socket (cont'd)

```
class ServiceThread extends Thread {
    Socket s;
    public ServiceThread( Socket p ) { s = p; }

    public void run( ) {
        ObjectInputStream ois = new ObjectInputStream( s.getInputStream());
        String cmd = (String) ois.readObject();
        if ( cmd.equals("register") )      { ... }
        else if ( cmd.equals("deregister") ) { ... }
        else if ( cmd.equals("bcast") )    { ... }
        else { System.err.println("Unknown command!"); }
        ois.close();
        s.close();
    }
}
```

Example: Chat client Socket

- 1) Registers to Chat server at known server port
 - (sends “register” + client IP addr + client port number)
- 2) Waits for messages on client port
 - (checks if starts with “msg” + displays message)
- 3) Sends messages to server
 - (sends message with “broadcast” prefix to server)

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

```
server.broadcast("Hello");
```

```
Socket s = new Socket(host, port);  
ObjectOutputStream oos = new  
    ObjectOutputStream( s.getOutputStream( ) );  
oos.writeObject( "broadcast" );  
oos.writeObject( "Hello" );  
oos.close();  
s.close();
```

Sockets

Serialization

Remote Objects

DISTRIBUTED PROGRAMMING

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

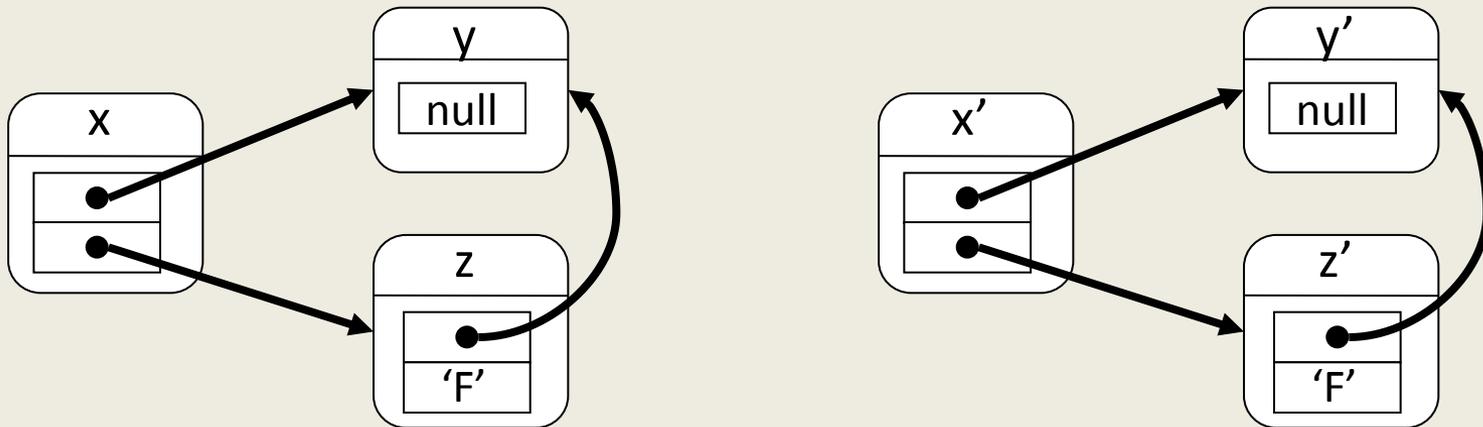
- Serialization needs access to private fields
 - Interface Serializable is used as tag
- Object streams serialize
 - Values of primitive types
 - Serializable objects
- All objects except strings are written only once

```
interface Serializable { }
```

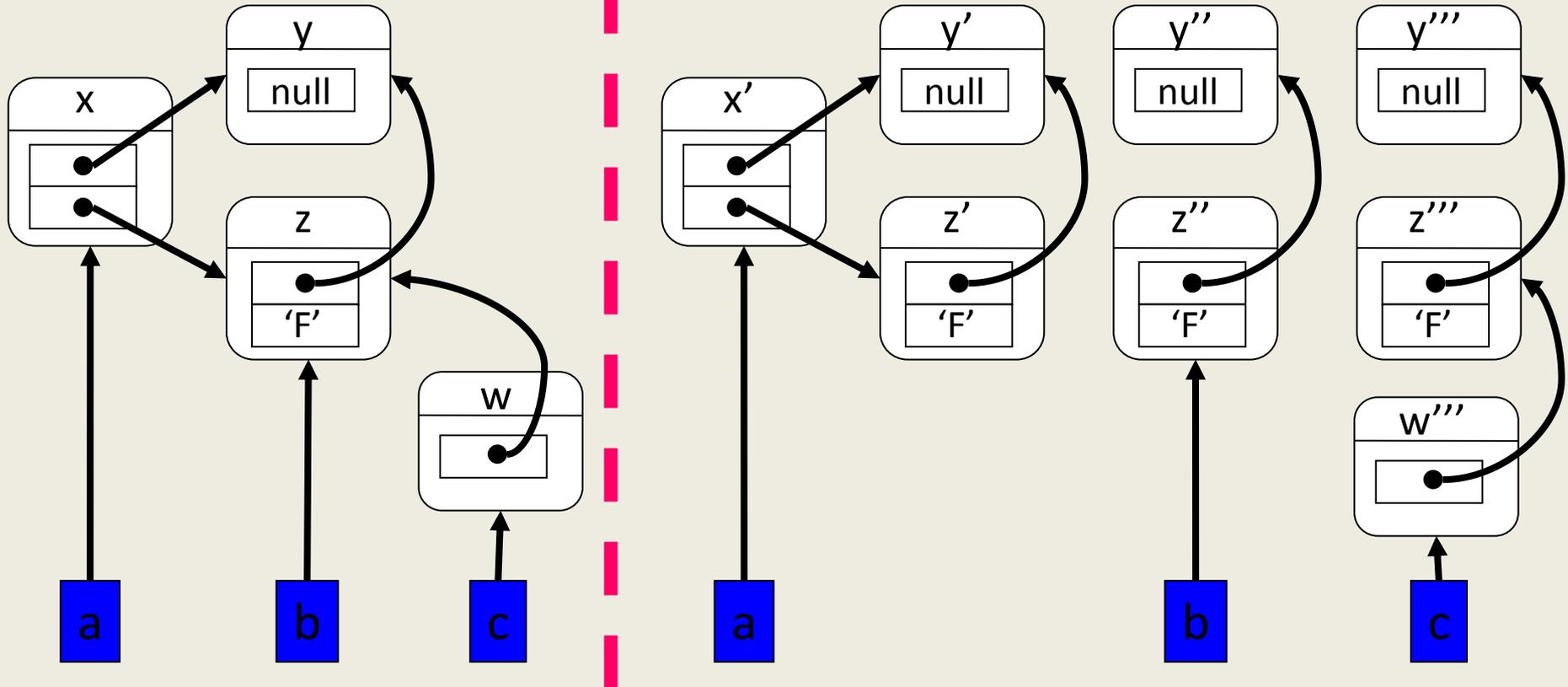
```
class ObjectOutputStream  
    extends OutputStream  
    implements ... {  
  
    void writeObject( Object obj )  
        throws IOException { ... }  
    ... }
```

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.

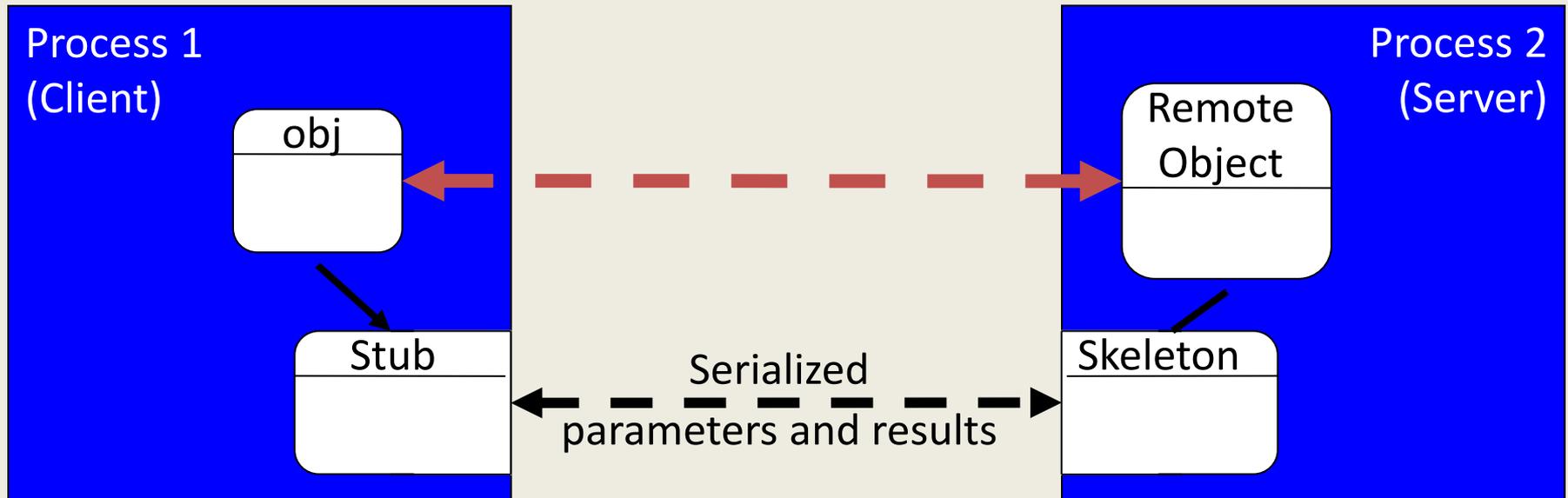
Sockets

Serialization

Remote Objects

DISTRIBUTED PROGRAMMING

Stubs and Skeletons



- Remote objects are represented locally by stubs.
- Stubs and skeletons provide communication.
- Code for stubs and skeletons is generated automatically.

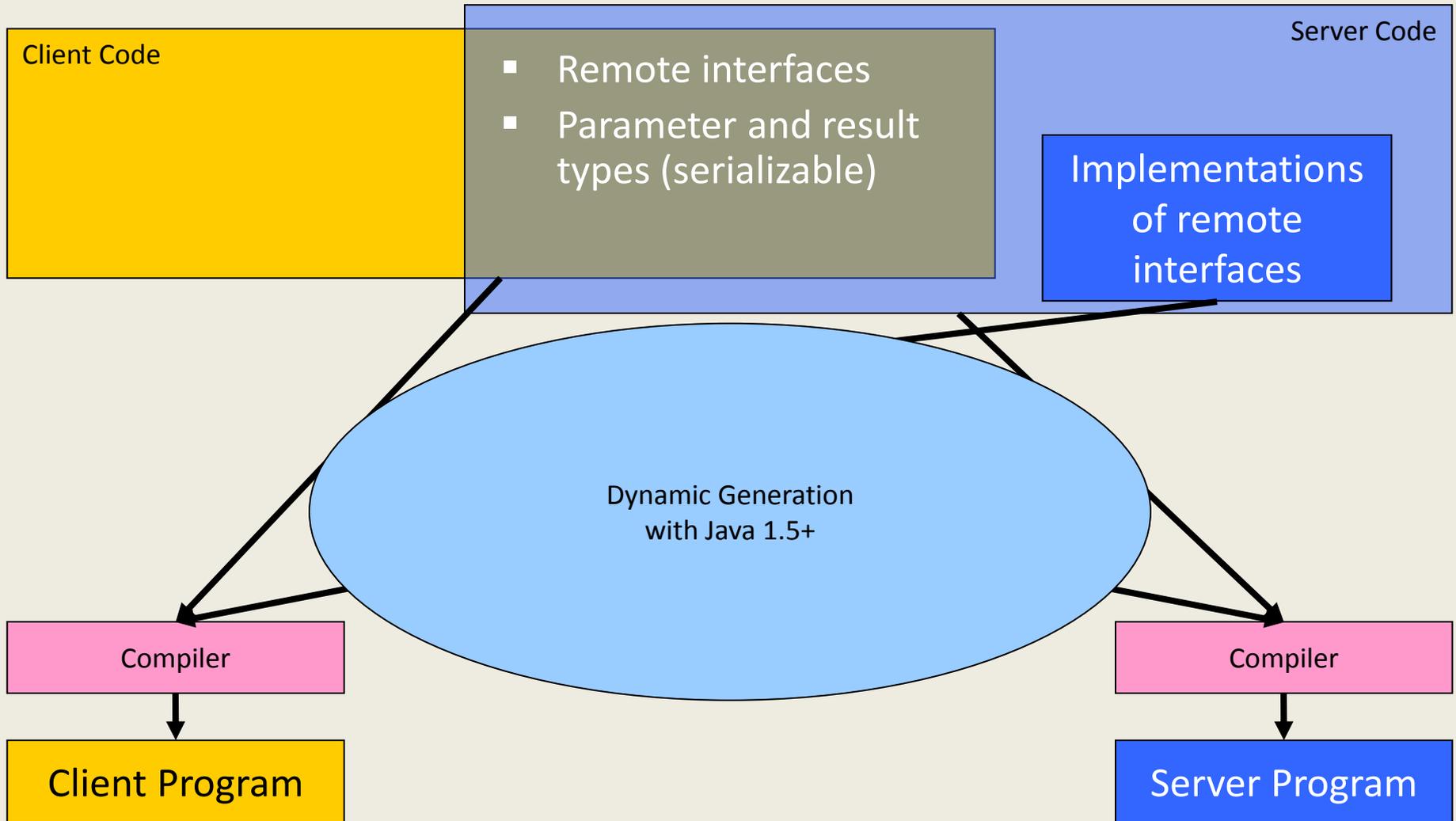
Remote Interfaces

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

```
interface Remote { }
```

```
interface ServerInterface extends Remote {  
    void register(ClientInterface c)  
        throws RemoteException;  
    void sendMessage(String msg)  
        throws RemoteException;  
}
```

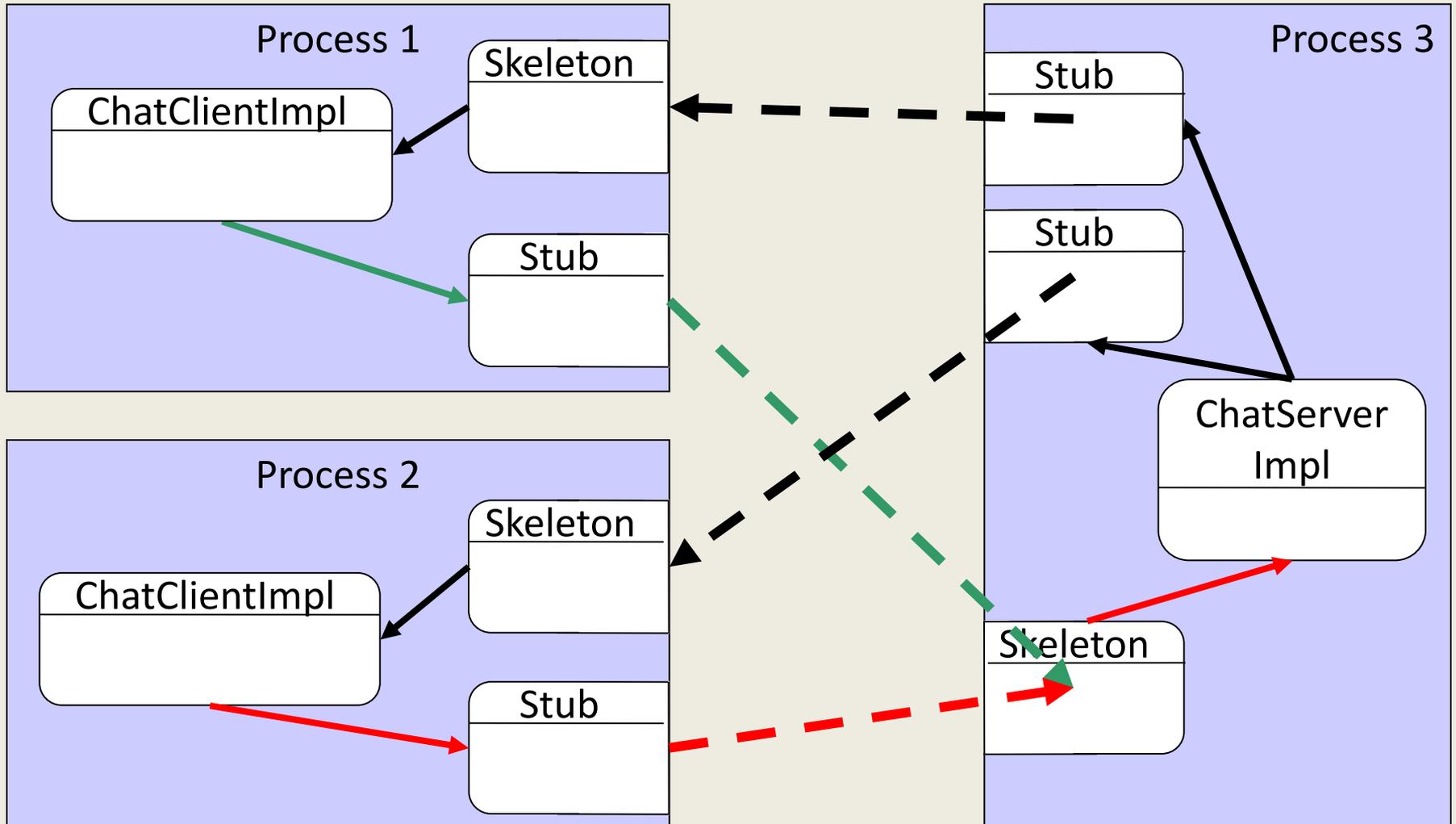
Programming with Remote Objects



Remote Method Invocation

- Implementations of remote objects extend `UnicastRemoteObject` (or similar classes).
- Constructors may throw exception.
- 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.

Process Interaction of Chat application



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.

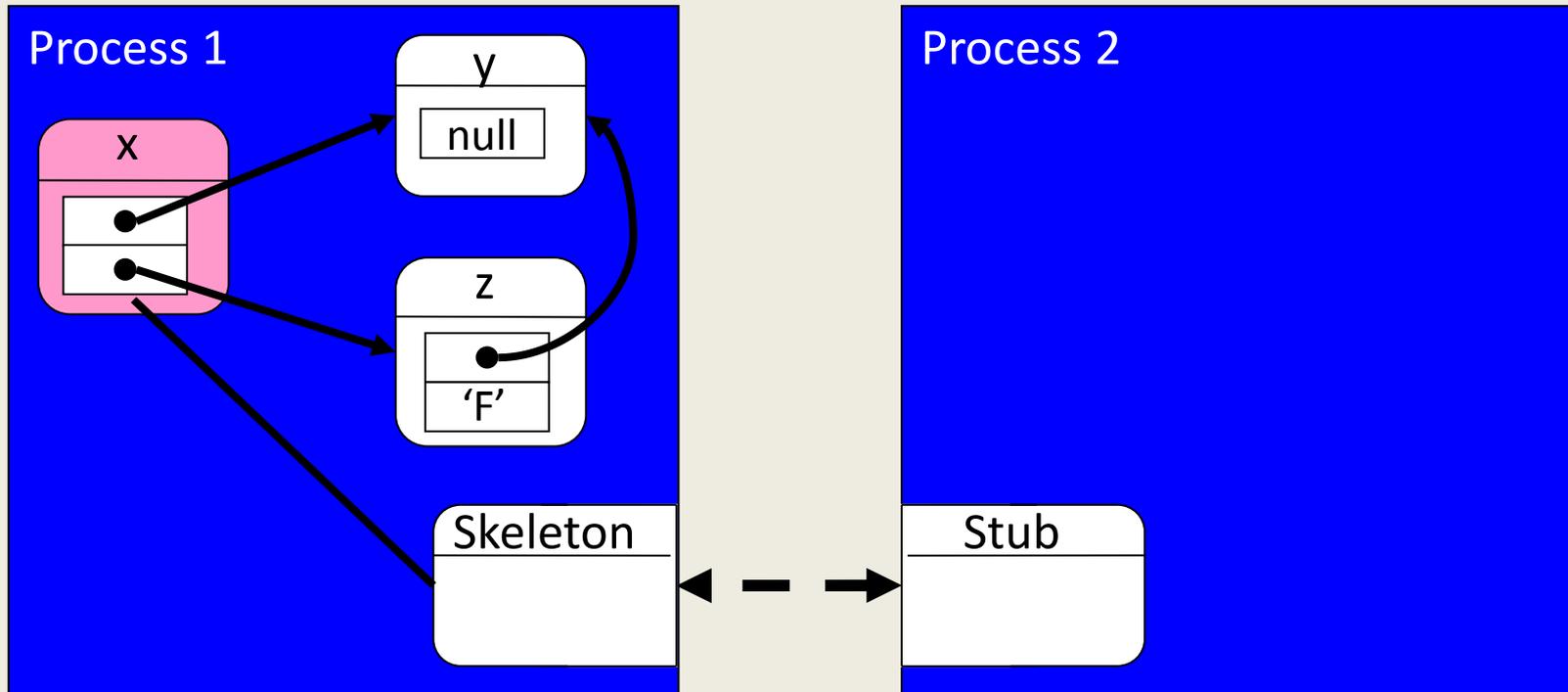
```
class Naming {  
    static Remote lookup( String name )    throws ... { ... }  
    static void rebind( String name, Remote obj ) throws ... { ... }  
    ...  
}
```

Using RMI in Java

1. Define **interface of remote object** (extends Remote)
2. Define **implementation of remote object** (extends UnicastRemoteObject)
3. Start **name server** (rmiregistry)
4. Server program **registers remote objects** at registry
5. 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



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

Further references

- Slides of *Konzepte objektorientierter Programmierung*:
<http://pm.ethz.ch/teaching/as2008/KOOP>
- Sun's RMI Tutorial:
<http://download.oracle.com/javase/tutorial/rmi/index.html>
- URLClassLoader (load classes from given URLs):
<http://download.oracle.com/javase/6/docs/api/java/net/URLClassLoader.html>
- Reflection (examine and manipulate running program)
<http://download.oracle.com/javase/tutorial/reflect/index.html>

Task:

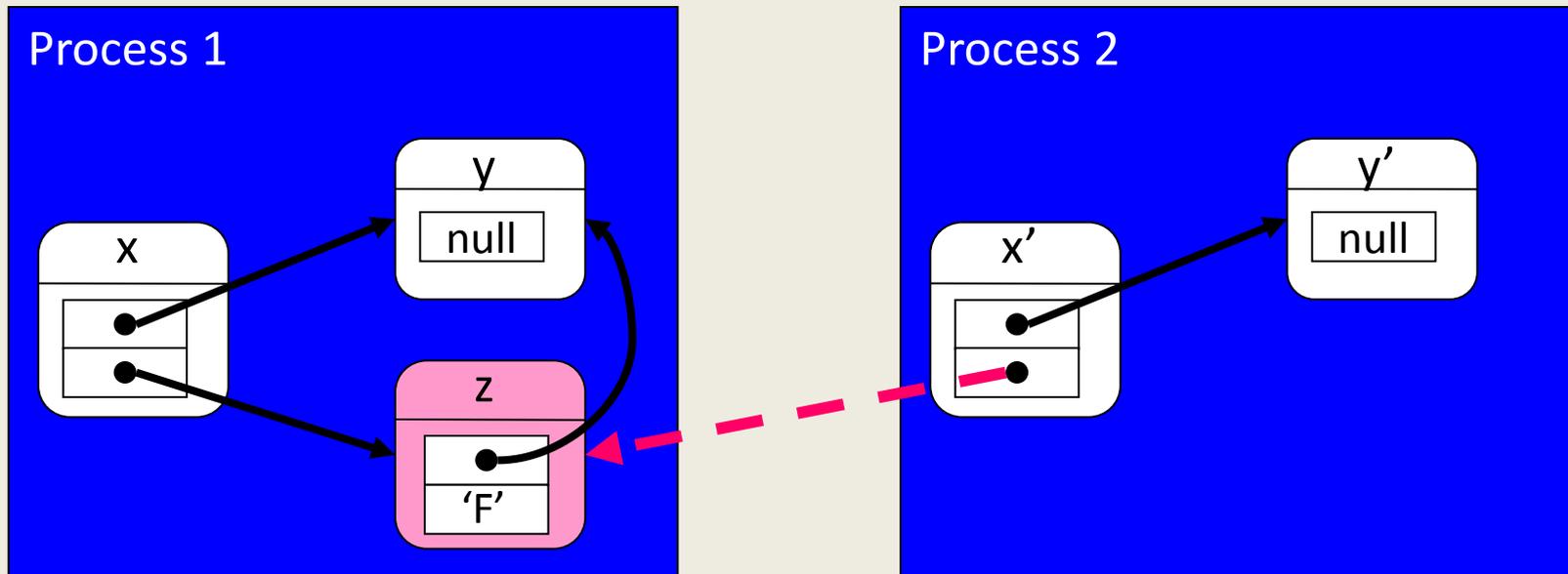
Implement Distributed Chat application

- Create Remote interfaces for server and client side.
- *Server side:*
 - Register server on a given port (and localhost):
`LocateRegistry.createRegistry(PORT).bind(SERVICE, server)`
 - Implement method for registering new clients.
 - Implement method for broadcasting messages to registered clients.
- *Client side:*
 - Register application to server:
`LocateRegistry.getRegistry(PORT).lookup(SERVICE)`
 - Implement method for receiving broadcast messages.
 - Implement method for sending messages to server.

BACKUP

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

