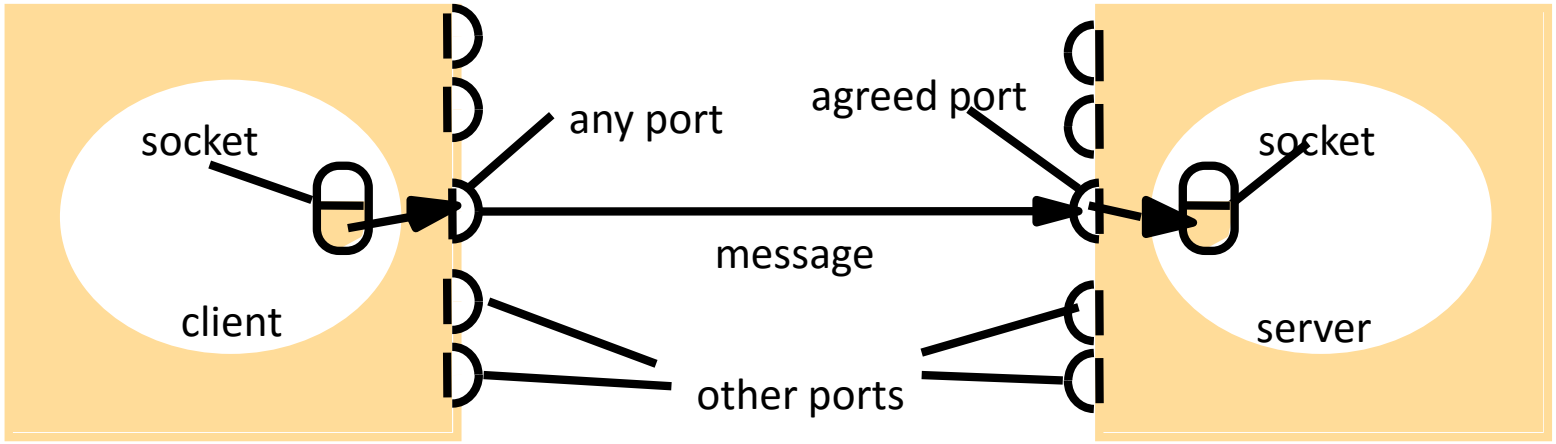
Distributed computing systems

Sockets and RPC

Direct message transmission: Sockets

Direct message transmission: Sockets

- Uses transport layer directly in the form of Middleware.



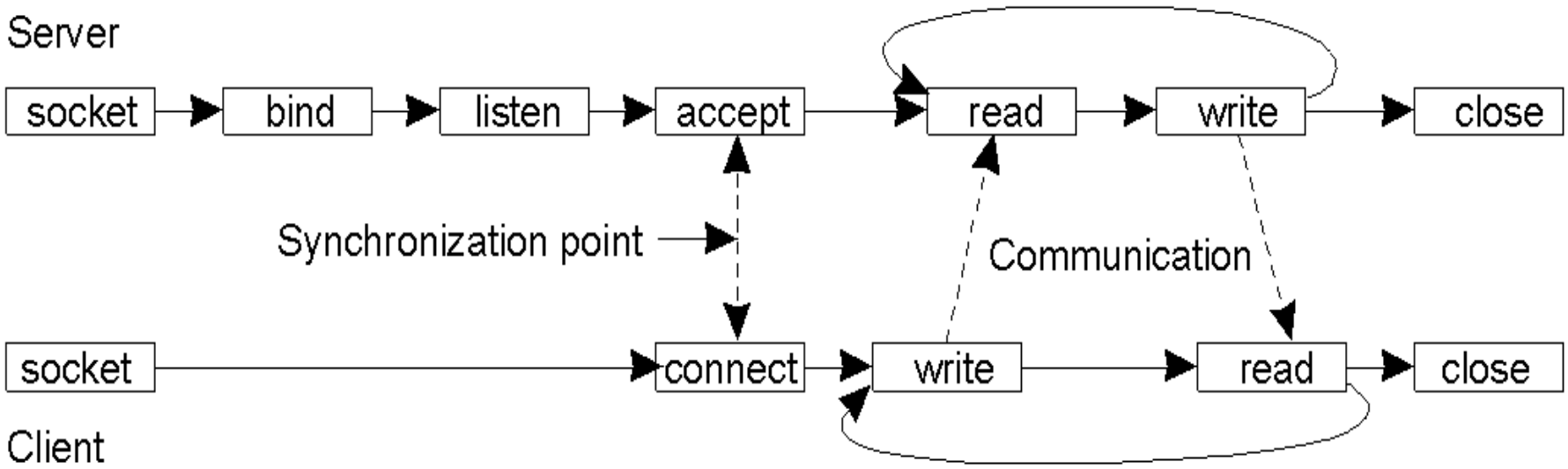
- **A socket** is an abstract object that represents the endpoint of the connection.
- TCP/IP socket is a combination of IP address and port number, for example, 10.10.10.10: 80.
- Socket interface first appeared in BSD Unix.

Berkeley Sockets API (1)

Socket primitives for TCP/IP.

Primitive	**Meaning**
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Berkeley Sockets (2)



Socket implementation example

C # supports two types of network connections:

- Server using the TcpListener class objects;
- the client implemented by using objects of the TcpClient class.

TcpListener and TcpClient objects

- An object of TcpListener class allows **only to listen** to a specific port on your computer.
- Any processes of data transmission via this socket are carried out using the TcpClient object.
- The AcceptTcpClient() method of the TcpListener class returns the TcpClient object that provides the listening port.

Server example

```
using System.Net;
using System.Net.Sockets;

Int32 port = 13000;

IPAddress localAddr = IPAddress. Parse
   ("127.0.0.1");

TcpListener server = new TcpListener (localAddr,
   port);

server.Start ();

//Start listening on port
TcpClient client = server.AcceptTcpClient ();
//After connection create message flow
NetworkStream stream = client.Getstream();
```

Messaging

Writing messages

```
Byte [] bytes = new Byte
  [256];

String data = "text";

bytes =
  System.Text.Encoding.UTF.
  GetBytes (data);

stream.Write (bytes, 0,
  bytes.Length);
```

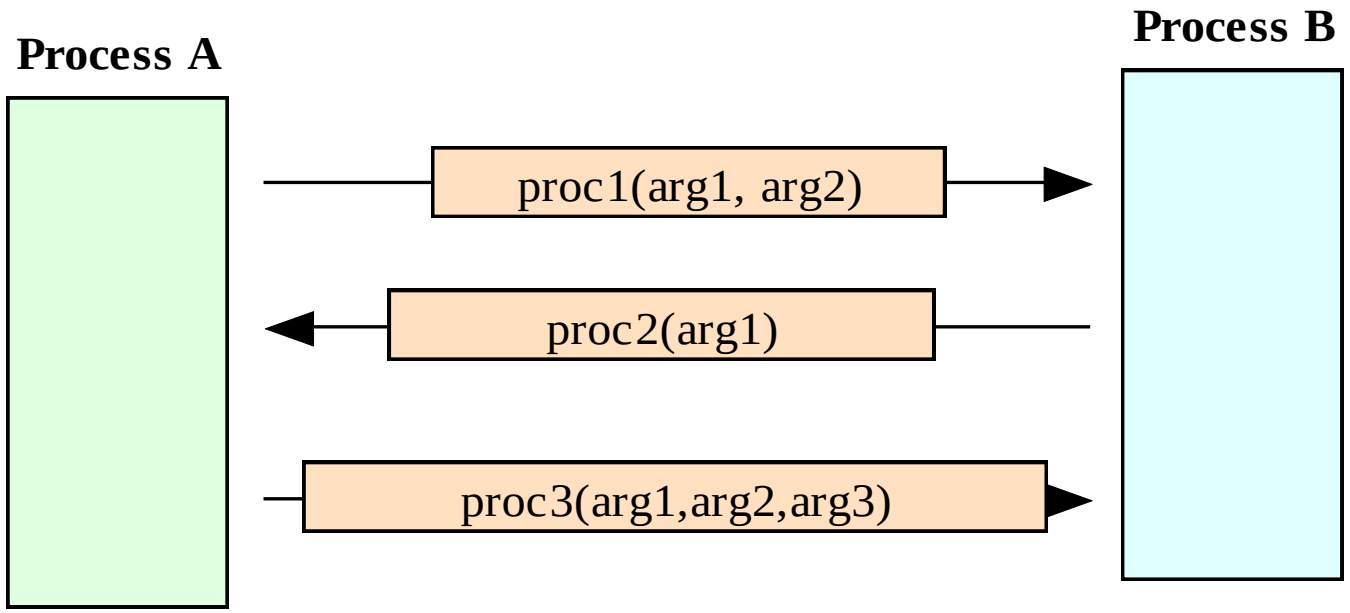
Reading messages

```
Byte [] bytes = new Byte
  [256];

String data = null;

int i = stream.Read (bytes,
  0, bytes.Length);


data = system.text.
  encoding.UTF8.GetString
  (bytes, 0, i);
```

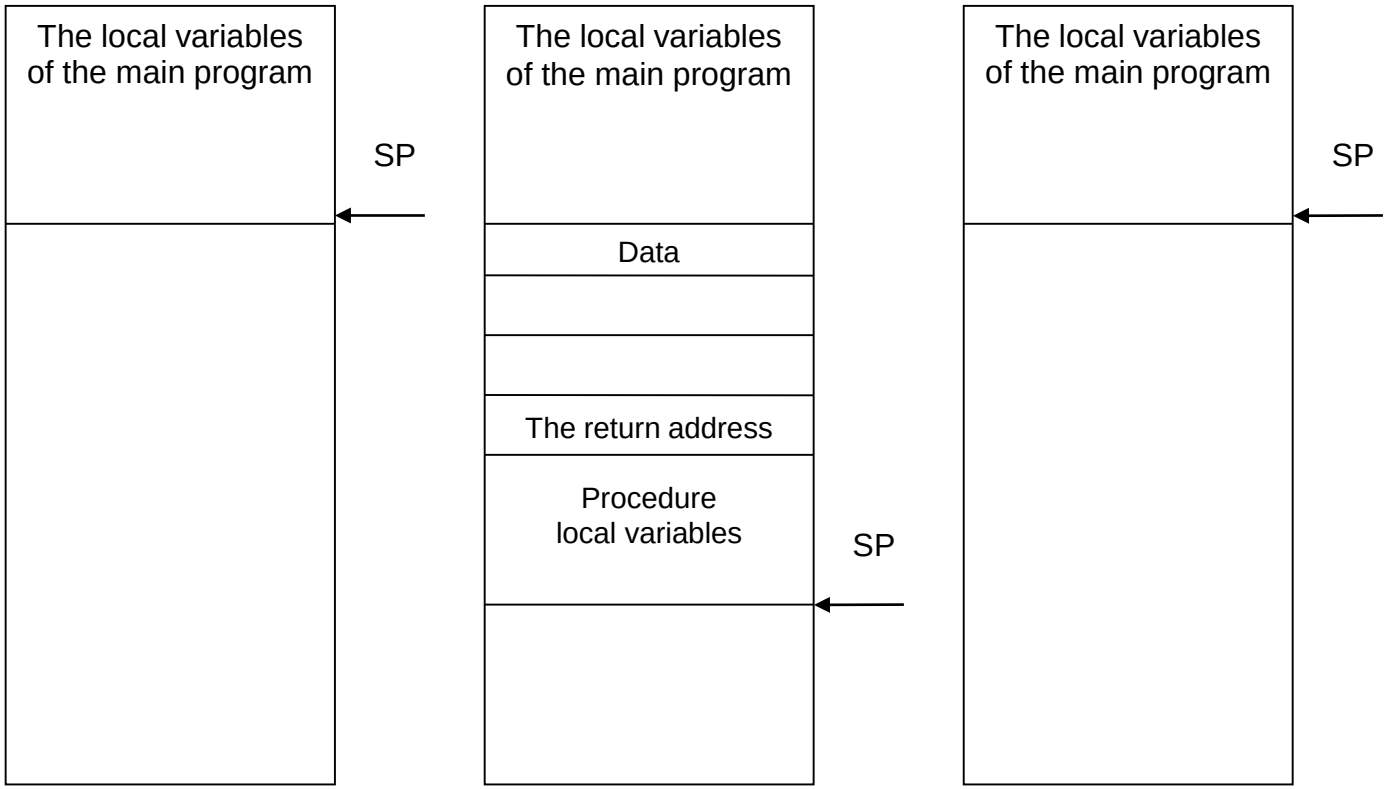
Remote procedure call
RPC-remote procedure call
RMI-remote method invocation

RPC Technology

- **Remote procedure call** is a technology that allows computer programs to call the function or procedure in a different address space.



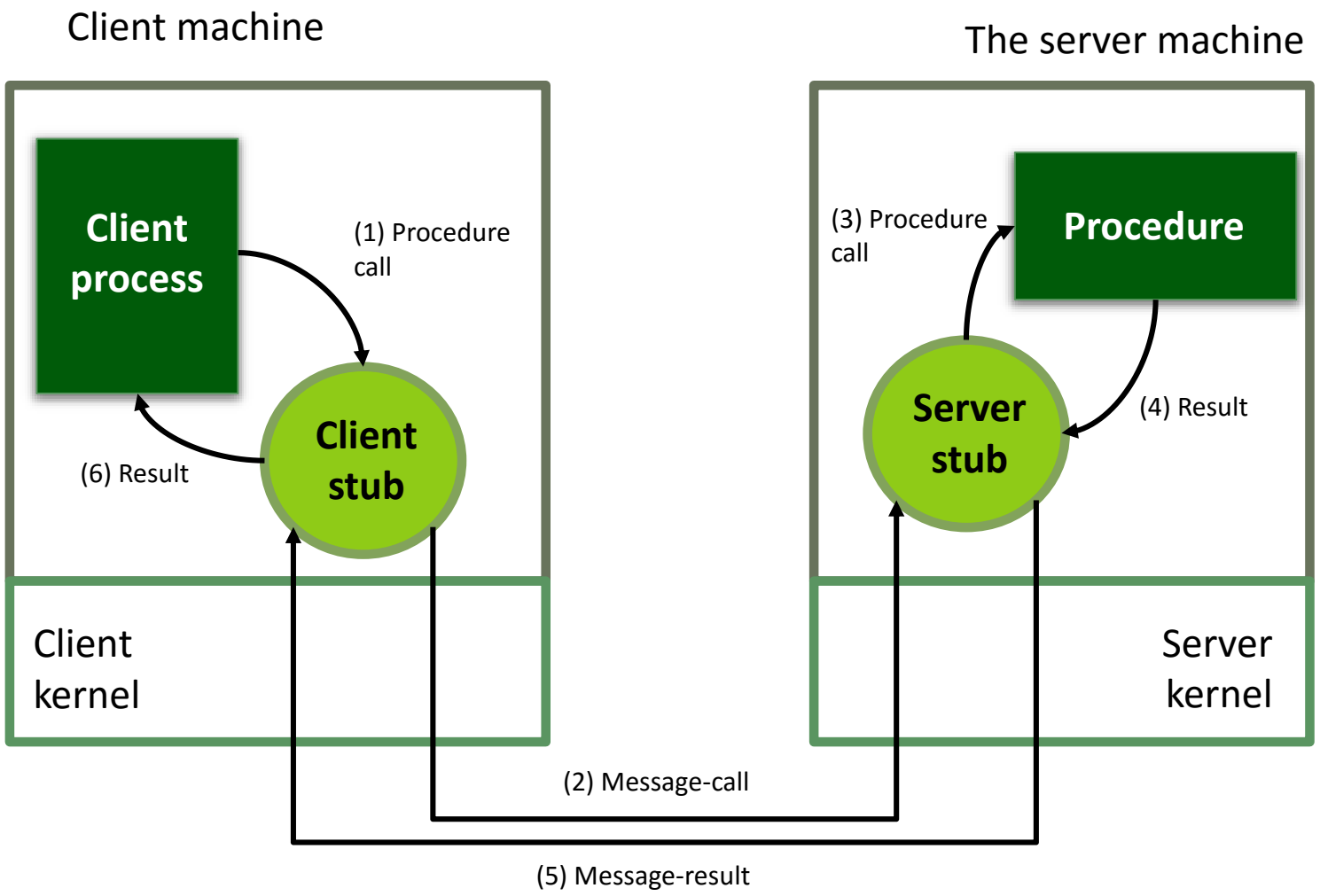
The stack when calling local procedures



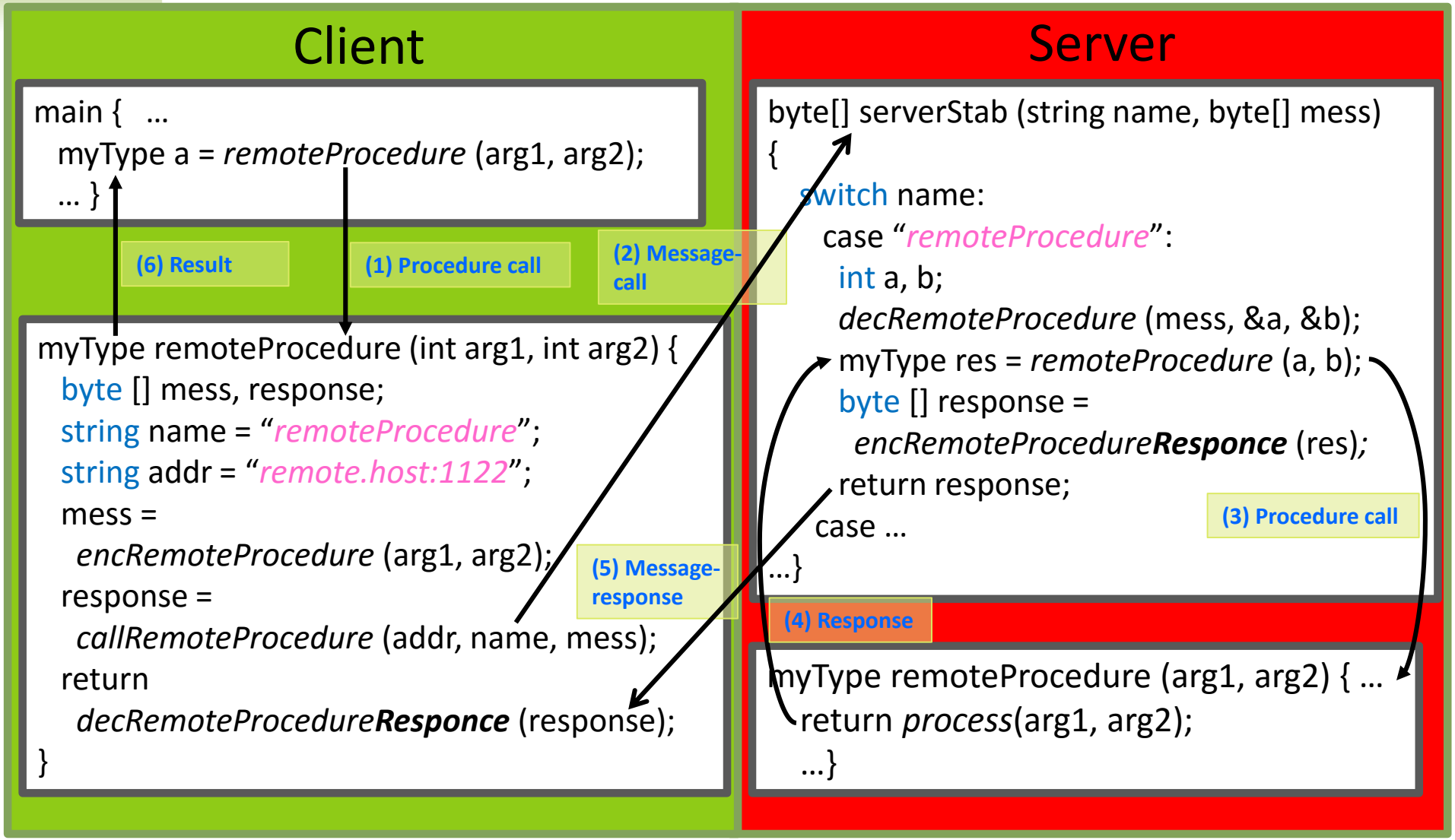
RPC Implementation

- The idea: remote procedure call "transparent" for the local process
- Instead of the local procedure we use the “client stub”.
- It is called as a local procedure, but instead of execution it sends a message the remote machine.

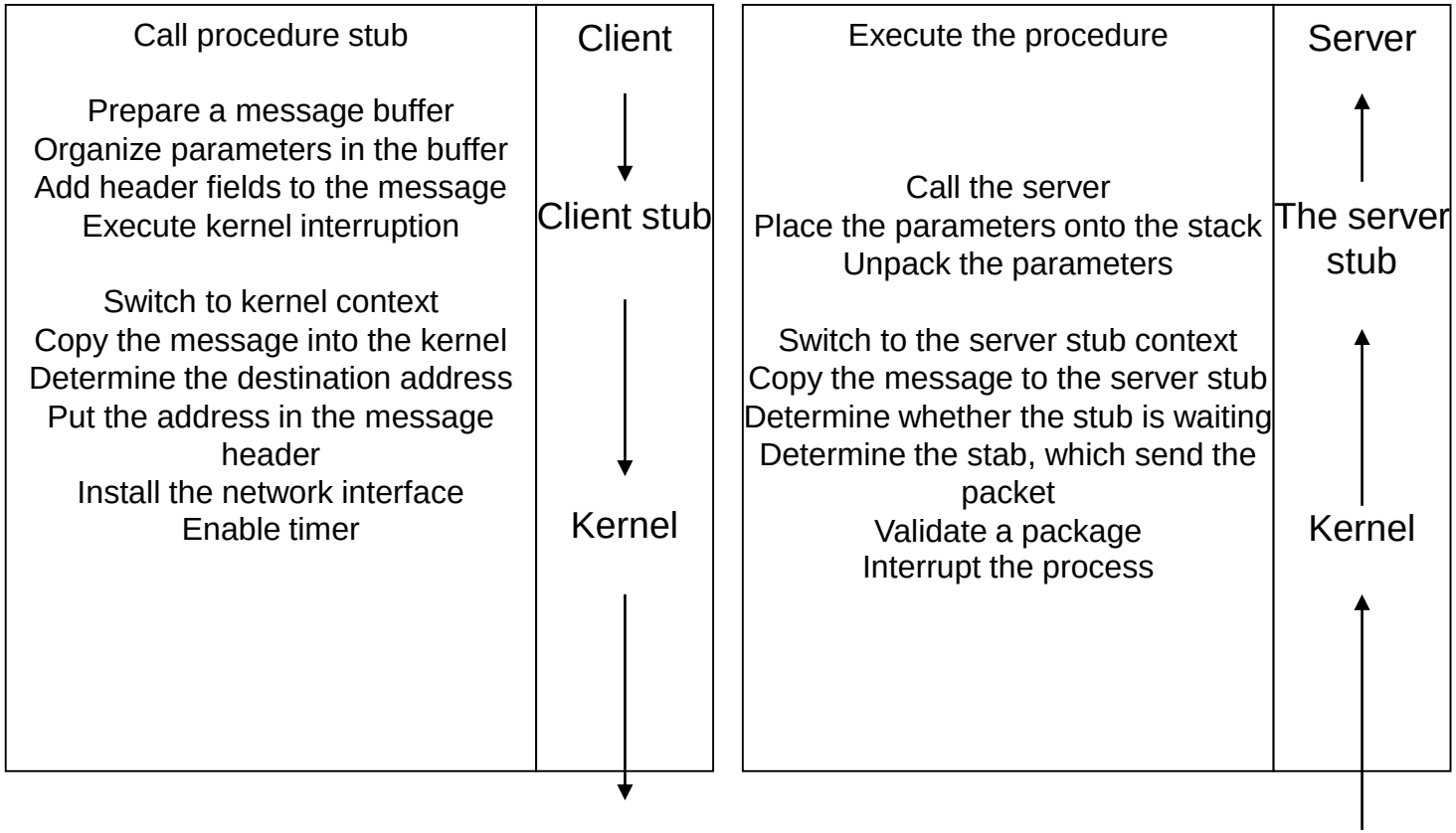
Remote Procedure Call



RPC Pseudo code



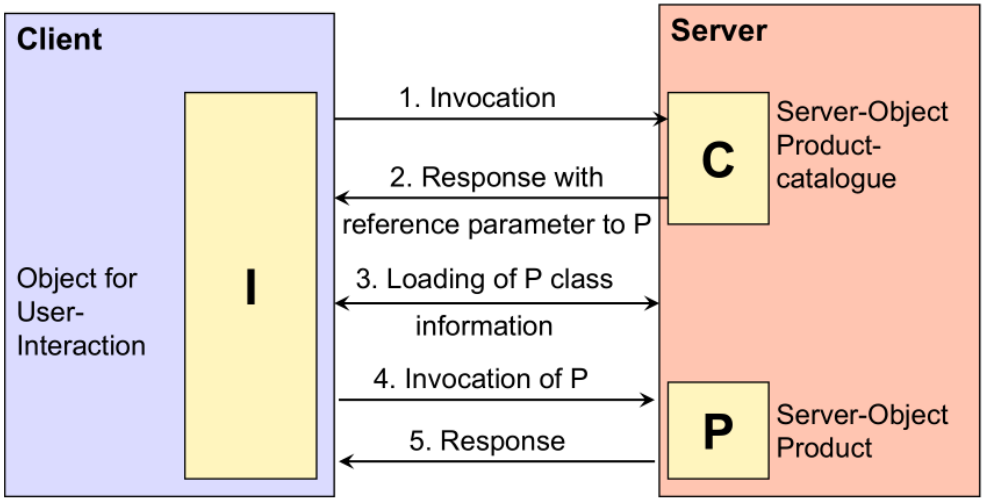
Stages of the RPC



Remote method invocation

In terms of OOP the Remote Method Invocation (RMI) concept was implemented.

- RMI allows to provide transparent access to the methods of remote objects, providing
  - delivery of parameters of the invoked method,
  - message to the remote object to execute the method
  - and the transfer of a return values back to the client

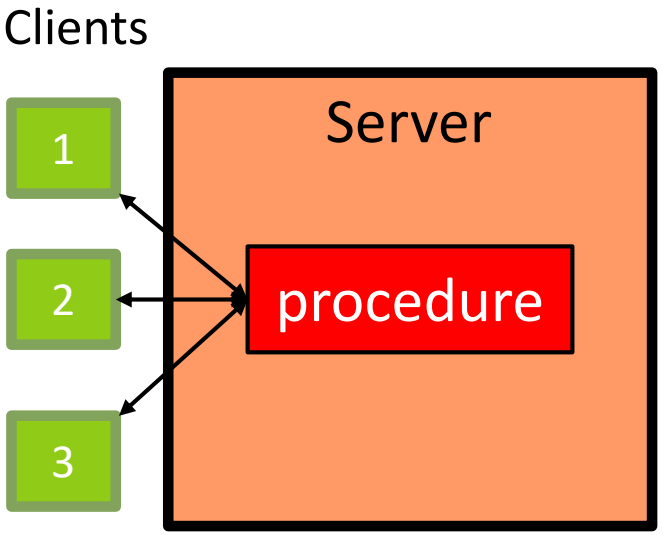


Remote object

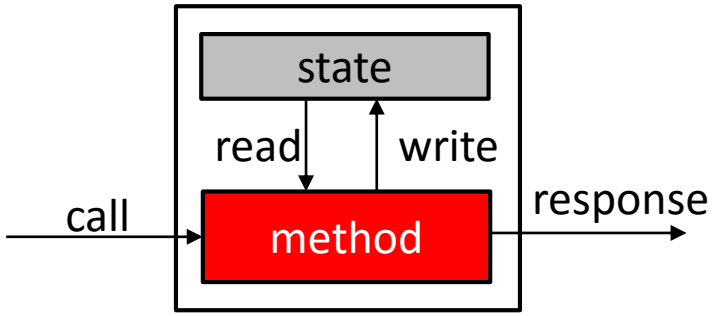
- The remote object is a collection of some data that determine its State. This State can be changed by calling some of his methods.
- Methods and fields of an object that can be used via remote calls, are available through the **external interface** of the objects class.

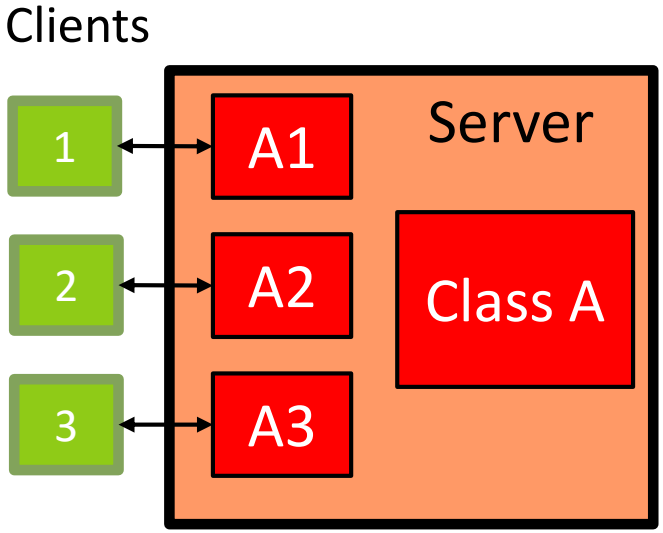
Remote Procedure VS Remote Object

Remote Procedure



Remote Object

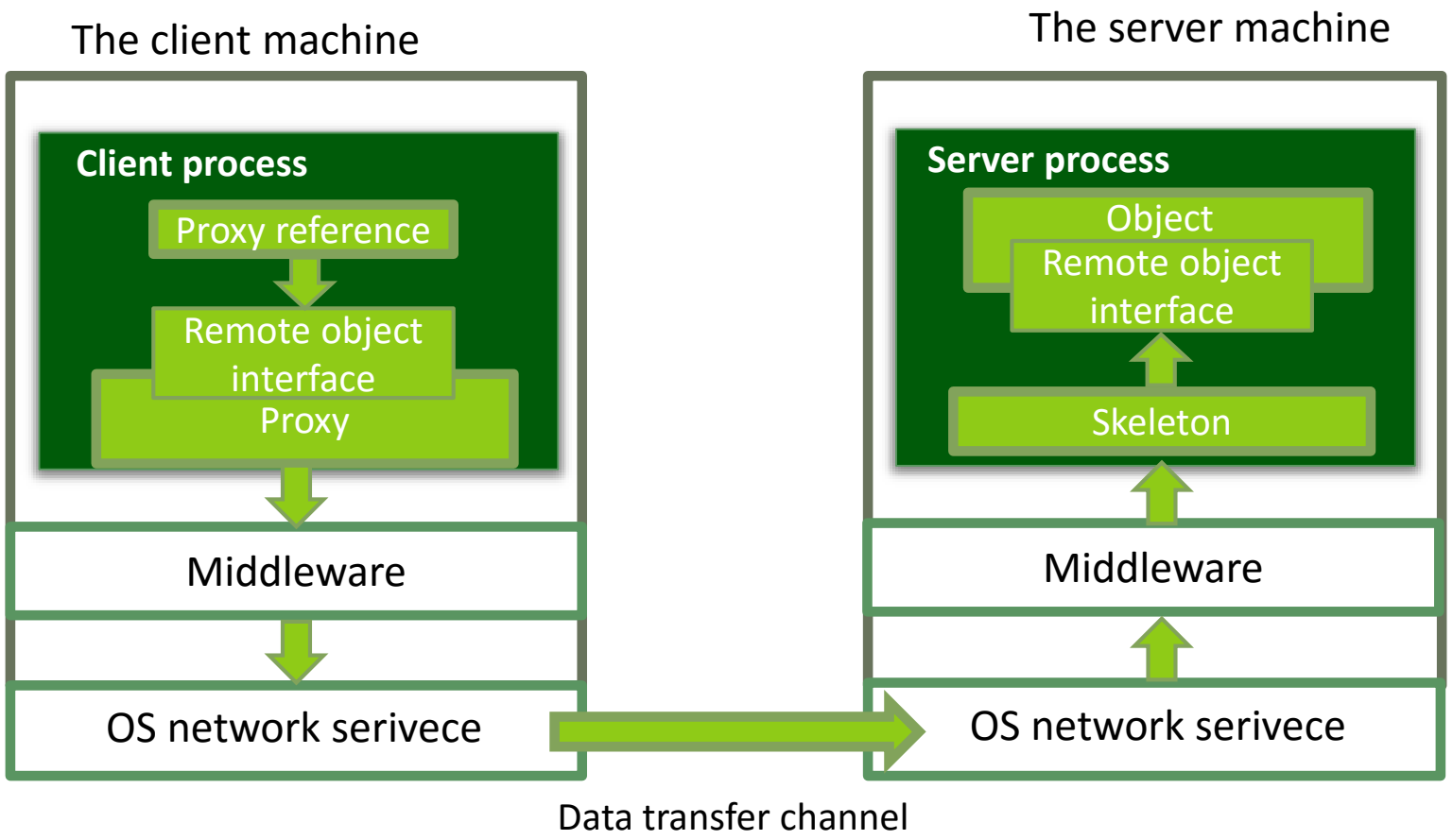




A Proxy and a Sceleton

- A client stub that invokes a remote object is called **the proxy**.
- **Proxy** implements the same interface as the remote object.
- The server-side stub is called **the skeleton** (in Java RMI)
- The skeleton is associated with a specific instance of the remote object and invokes the method with the desired settings

Remote object usage



Remote Method Invocation RMI

Interface:

```
public interface ProductCatalogue extends java.rmi.Remote
{
   ProductDescription[] searchProduct(String productType) throws java.rmi.RemoteException;
   Product provideProduct(ProductDescription d) throws java.rmi.RemoteException;
   int deleteProduct(ProductDescription d) throws java.rmi.RemoteException;
   int updateProduct(Product p) throws java.rmi.RemoteException;
   ...
}
```

Server – interface realization:

```
public class ProductCatalogueImpl extends java.rmi.server.UnicastRemoteObject
implements ProductCatalogue
{
   public ProductCatalogueImpl() throws java.rmi.RemoteException
   {   super();   }

   public ProductDescription[] searchProduct(String productType)
   throws java.rmi.RemoteException
   {
      ProductDescription[] desc = ProductCatalogue.getDescriptionByType(productType);
      return desc;
   }
    ...
```

Remote Method Invocation RMI

Server realization:

```
public class ProductCatalogueServer {
   public ProductCatalogueServer() {
      try {
         ProductCatalogue c = new ProductCatalogueImpl();
         Naming.rebind("rmi://localhost:1099/ProductCatalogueService", c);
      }
      catch (Exception e) {…}
   }
   public static void main(String args[]) {
      new ProductCatalogueServer();
   }
}
```

Client Realization:

```
public class ProductCatalogueClient {
   public static void main(String[] args)
   {
      try {
         ProductCatalogue c= (ProductCatalogue)Naming.lookup(
            "rmi://hostname/ProductCatalogueService");
         System.out.println( c.searchProduct("book");
      }
catch (Exception e) {…}
   }
}
```