COMMUNICATION

Contd.

CS435 Distributed Systems

Basit Qureshi PhD, FHEA, SMIEEE, MACM

https://www.drbasit.org/

TOPICS

- Remote Procedure Calls (RPCs)
- Encoding messages
- ONC (Sun) RPC
- Microsoft DCOM/COM+
- Java RMI
- Python RPyC and xmlrpc
- RPC in a nutshell

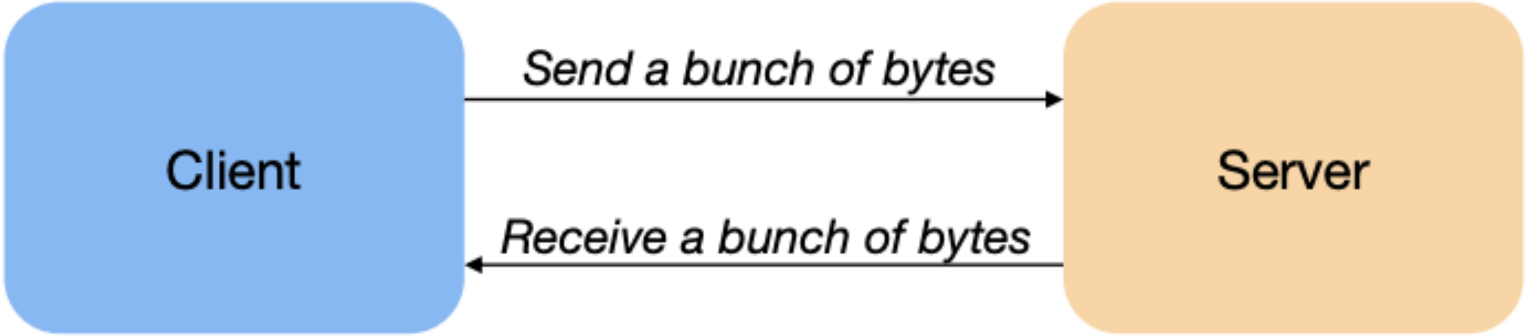
REMOTE PROCEDURE CALLS (RPC)

PROBLEM WITH SOCKETS

Socket interface forces a read/write mechanism

You have to implement Read and Write stream for TCP/UDP Sockets

- Client Sends a bunch of bytes to Server (Write)
- Server reads the bytes (Read)
- Server writes the bytes to the Client (Write)
- Client reads the bytes (Read)



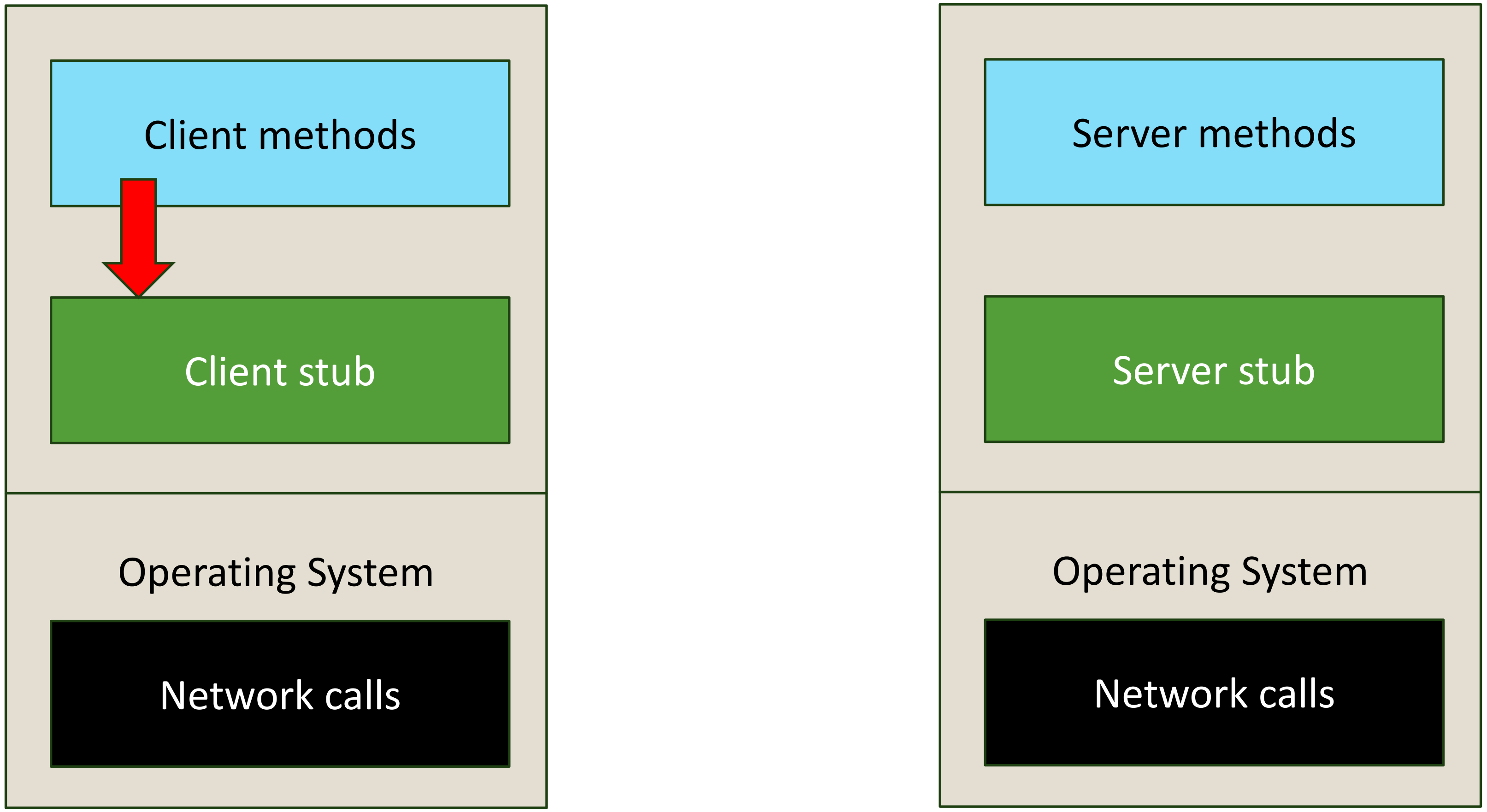
Is there a better option??

REMOTE PROCEDURE CALLS (RPC)

- 1984: Birrell & Nelson
- RPC: Allow programs to call procedures located on other machines
  - Conceal communication
  - No message passing at all is visible to the programmer.
- How?
  - Stub functions!
  - Gives the illusion (simulation) to the user that the call in local

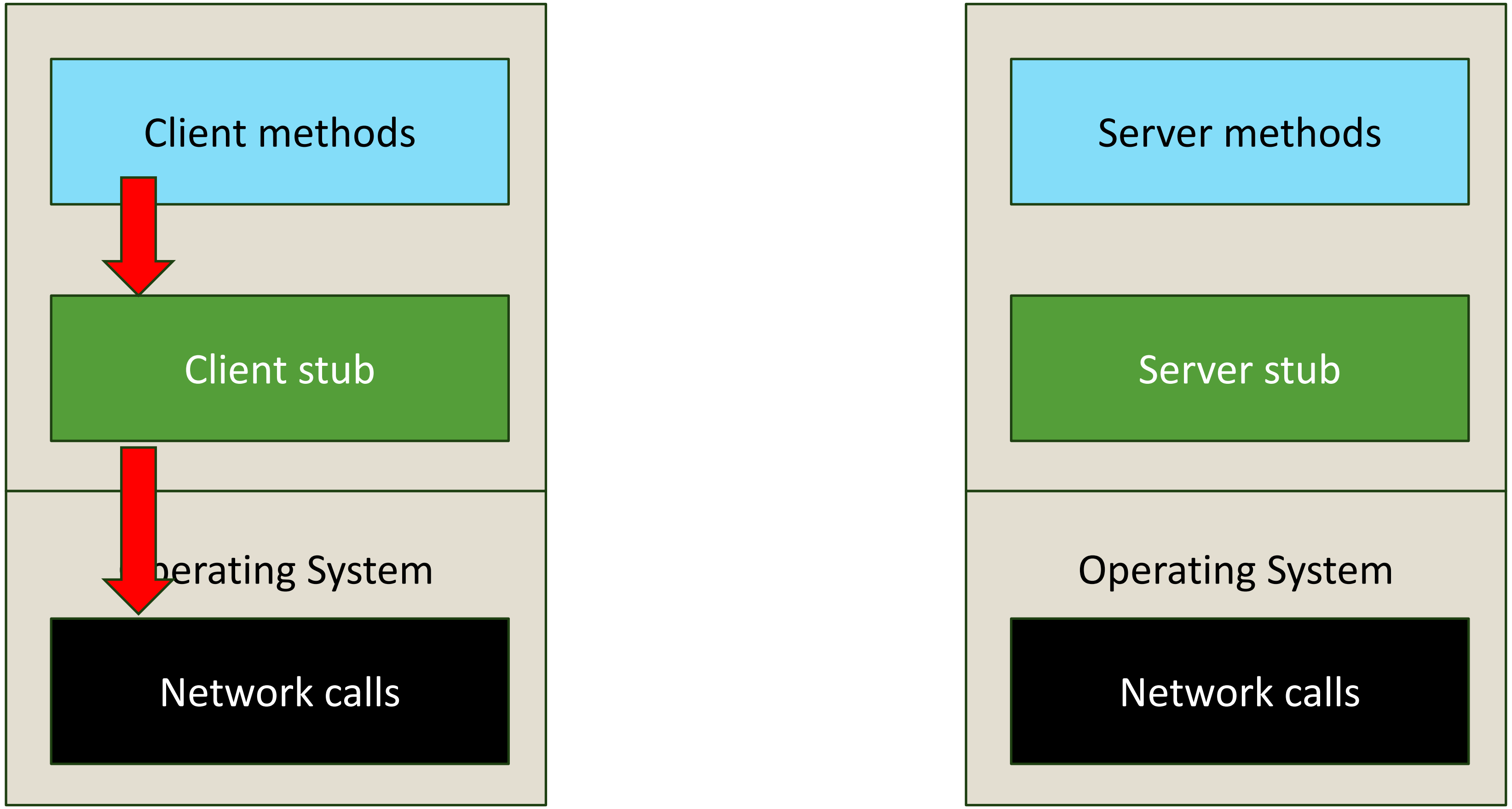
REMOTE PROCEDURE CALLS (RPC)

- 1. Client **calls** stub (parameters on stack)



REMOTE PROCEDURE CALLS (RPC)

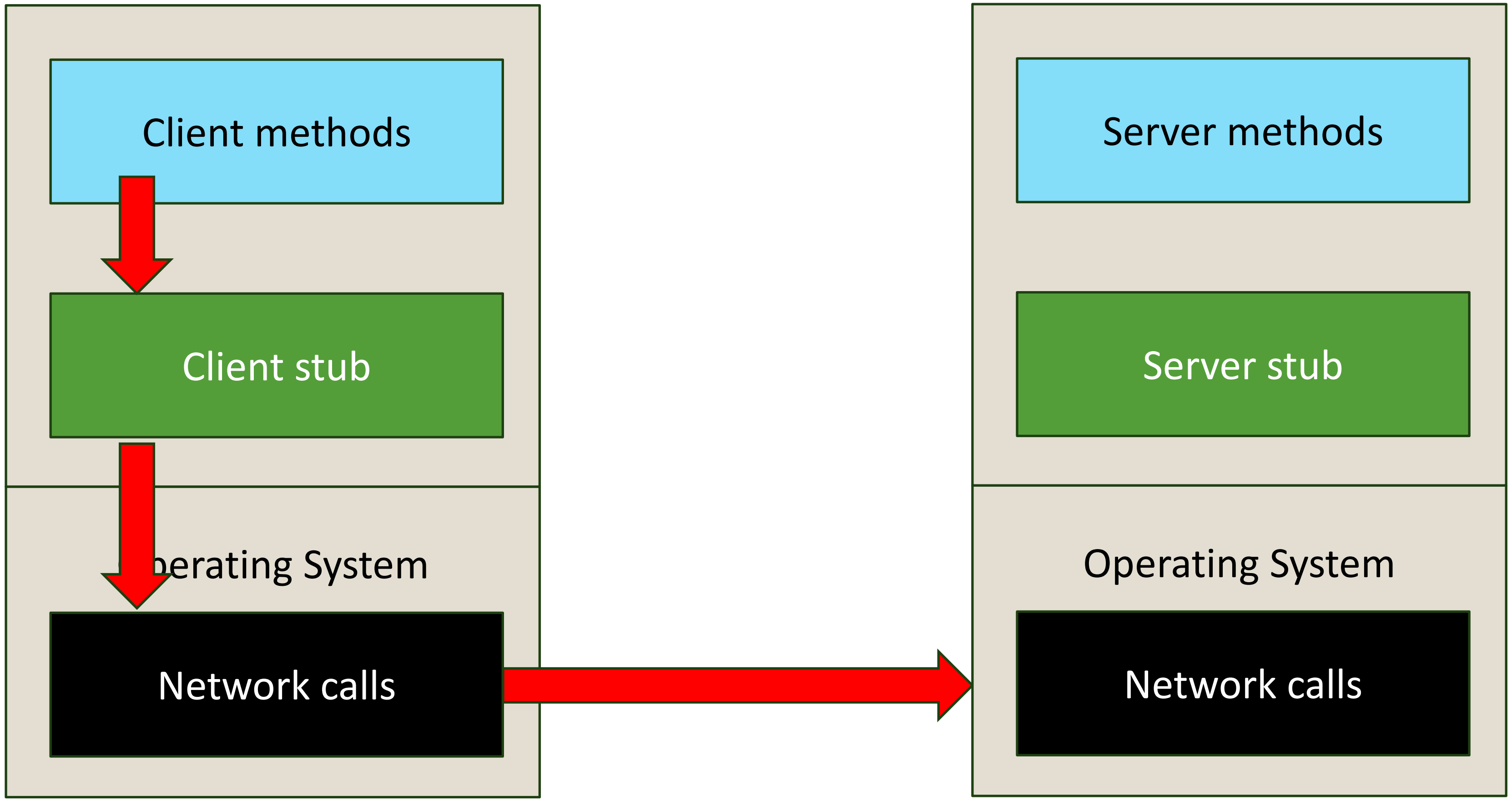
- 2. Stub **marshals** parameters to network



Marshal:
1. Set parameter for transmission over the network
2. Serialize messages
3. Initialize messages (method, objects, version etc)

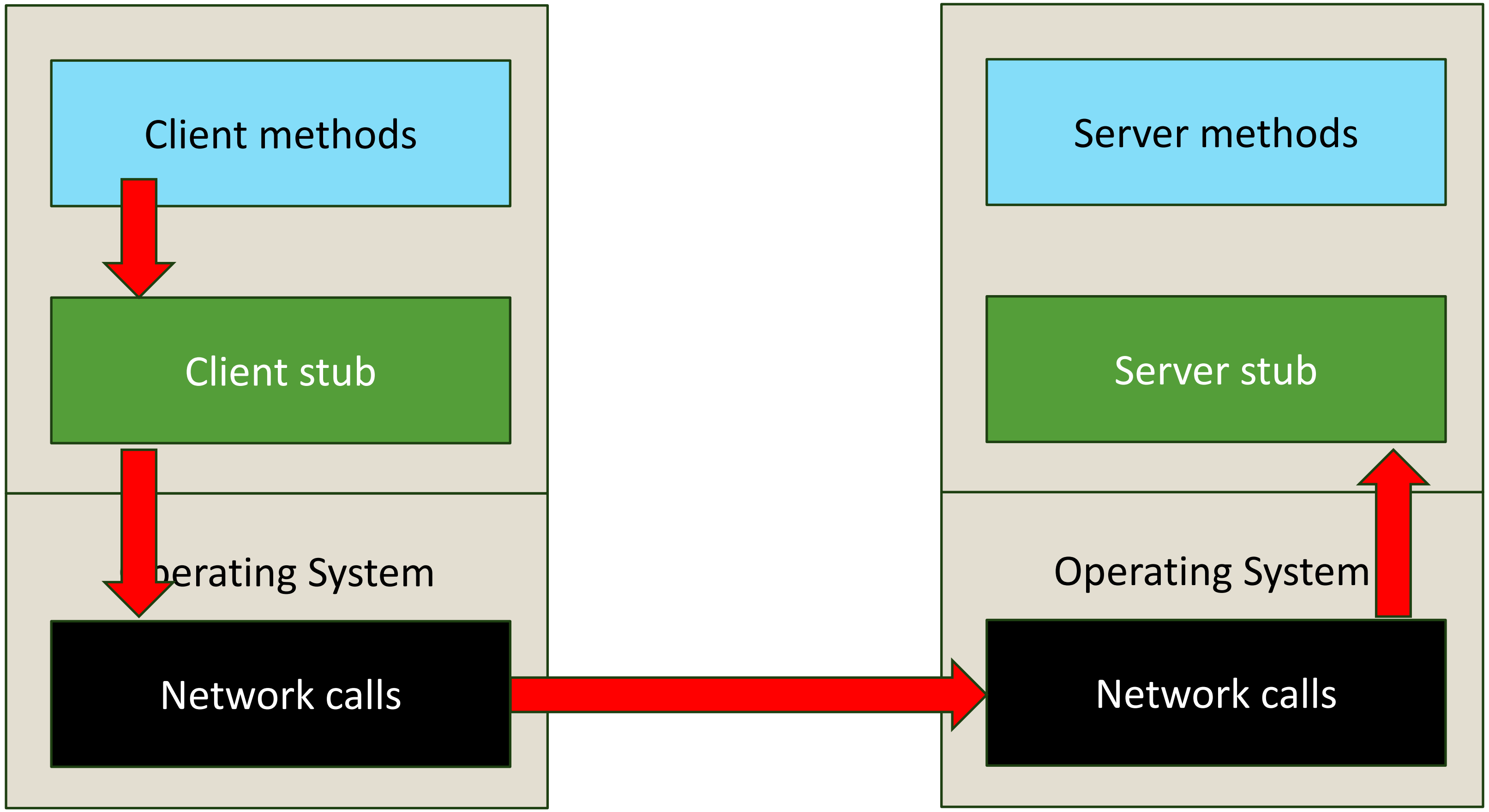
REMOTE PROCEDURE CALLS (RPC)

- 3. Message sent to server over the Network.



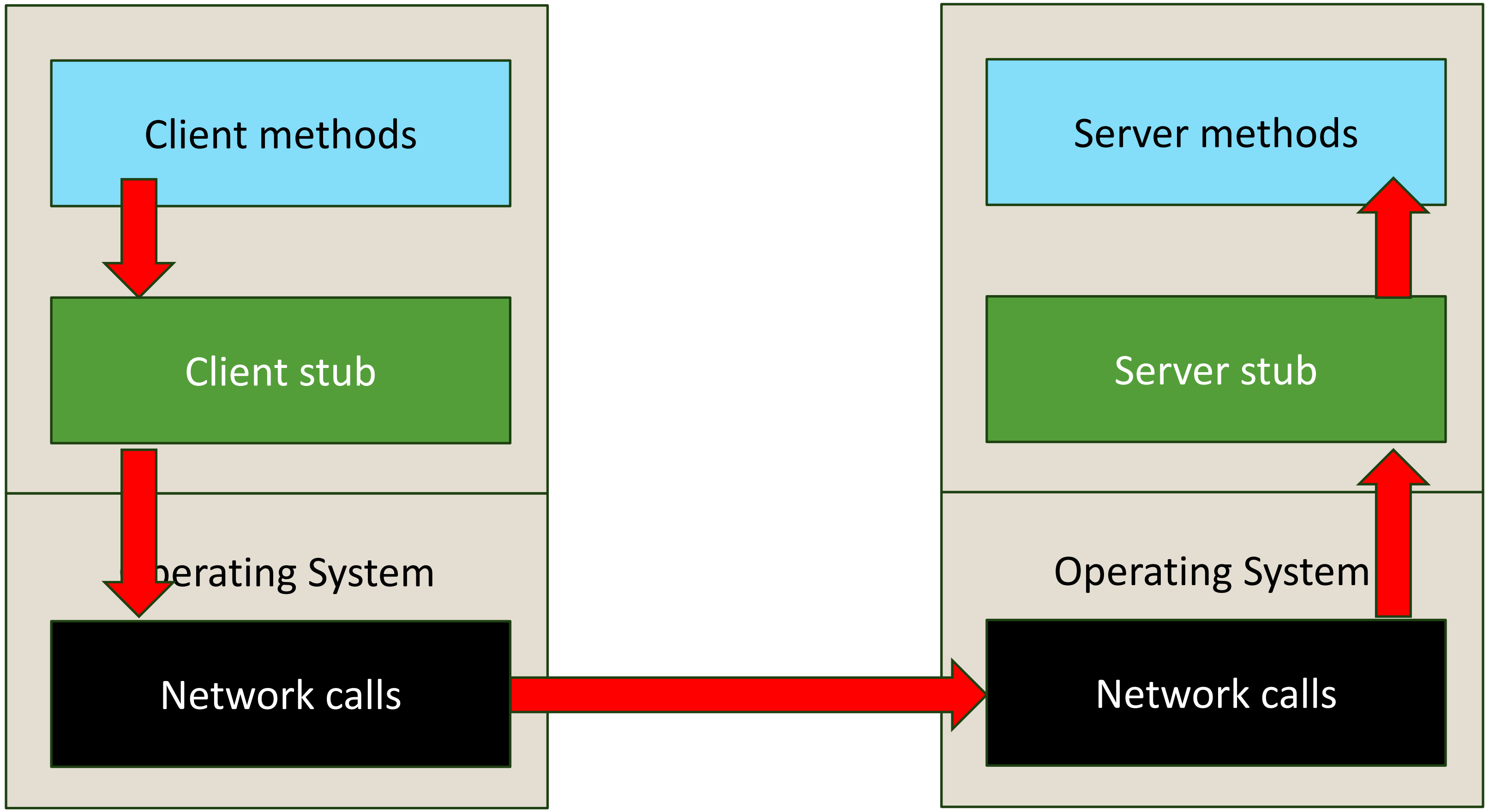
REMOTE PROCEDURE CALLS (RPC)

- 4. Message received, sent to the server stub



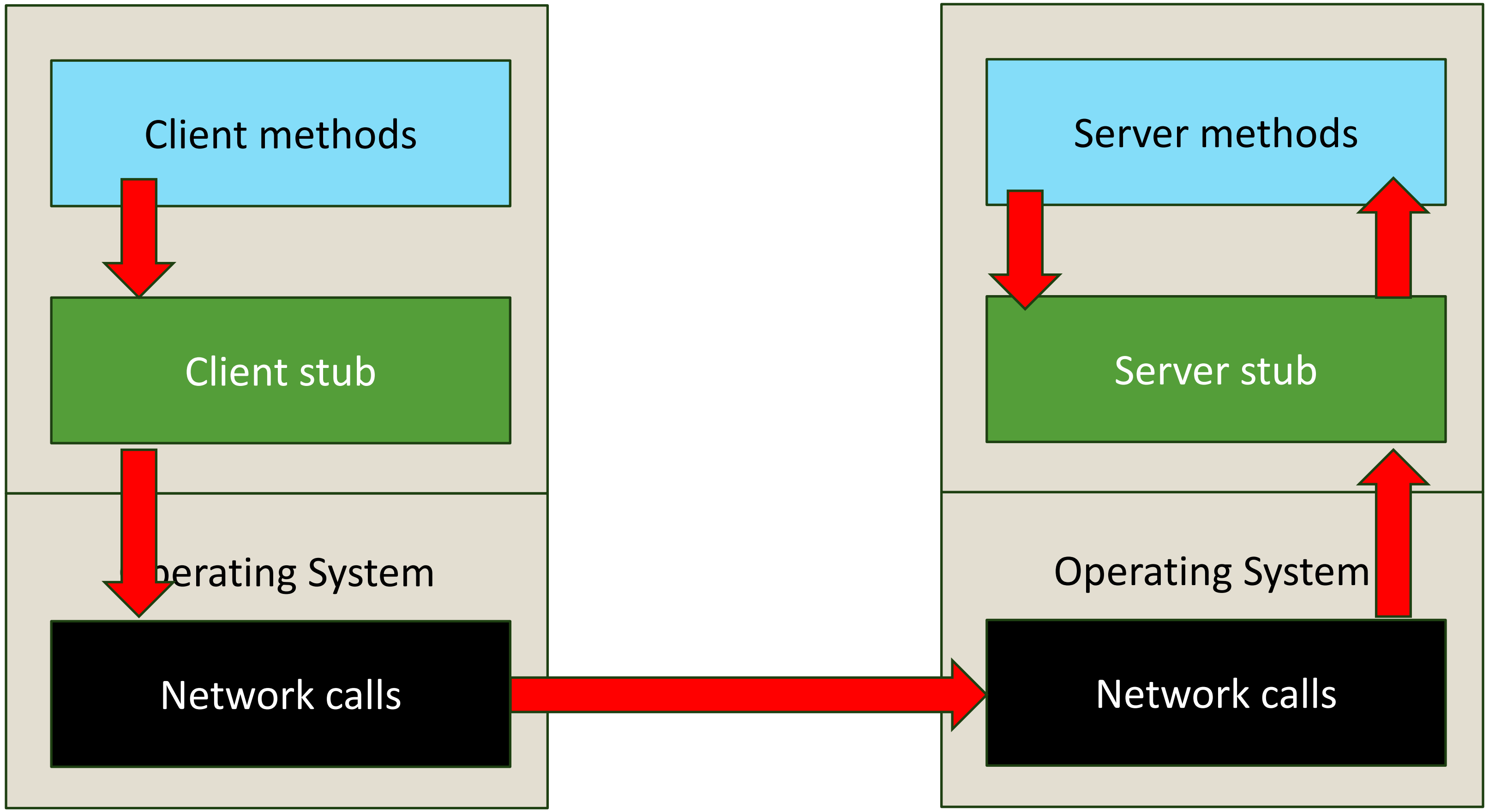
REMOTE PROCEDURE CALLS (RPC)

- 5. Unmarshal parameters, call server methods



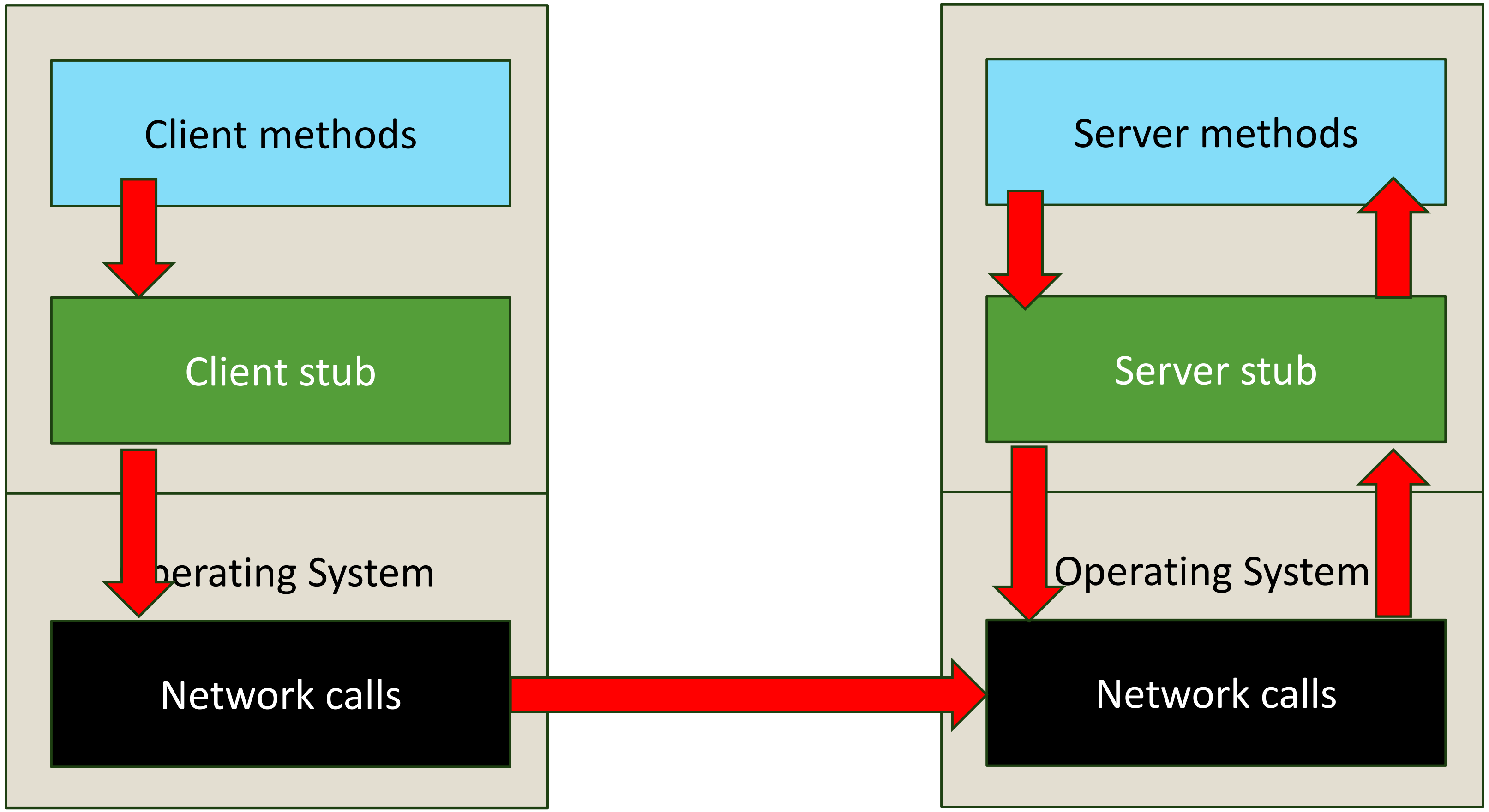
REMOTE PROCEDURE CALLS (RPC)

- 6. return from server methods



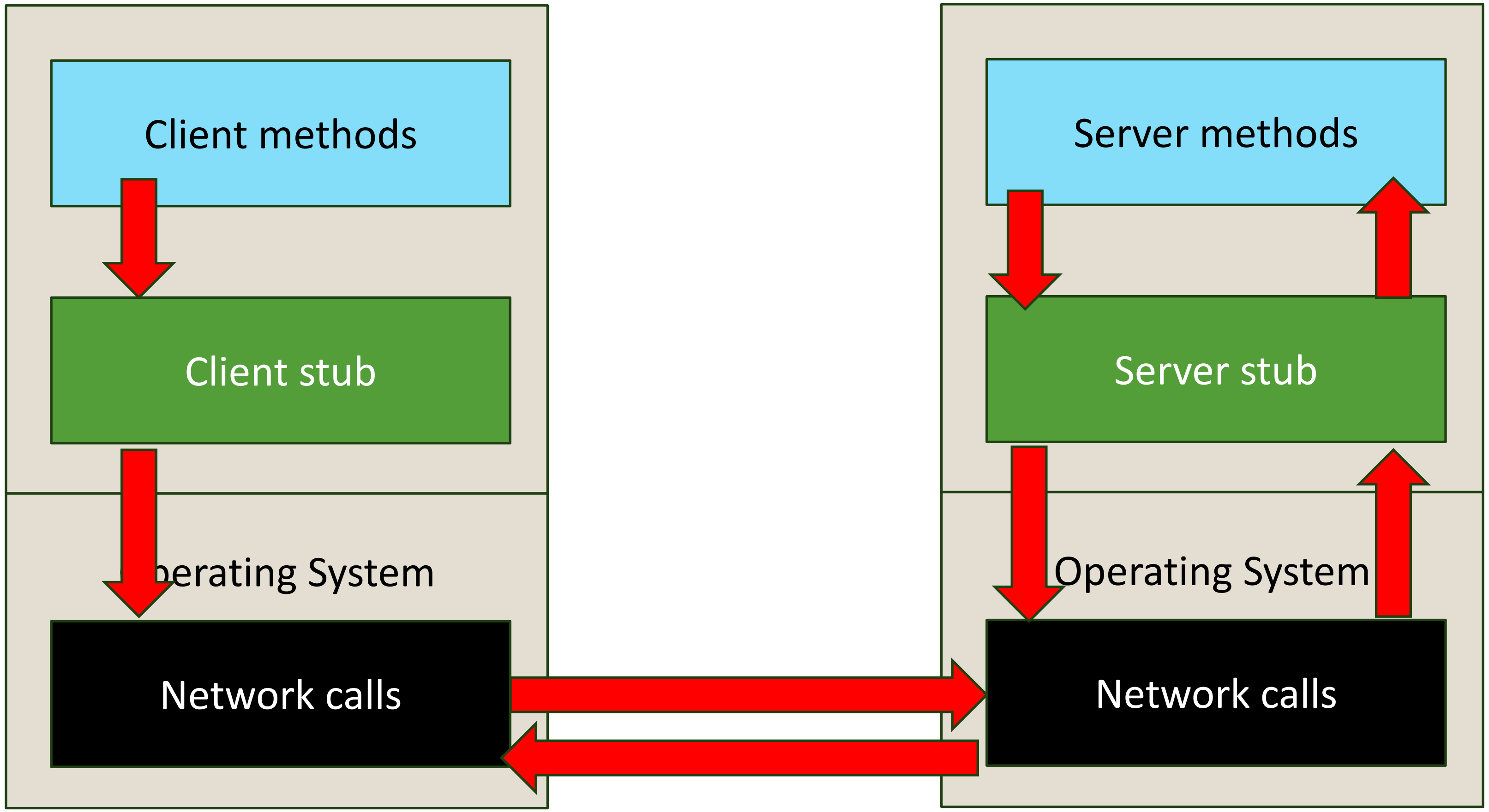
REMOTE PROCEDURE CALLS (RPC)

- 7. Marshal return value and send message



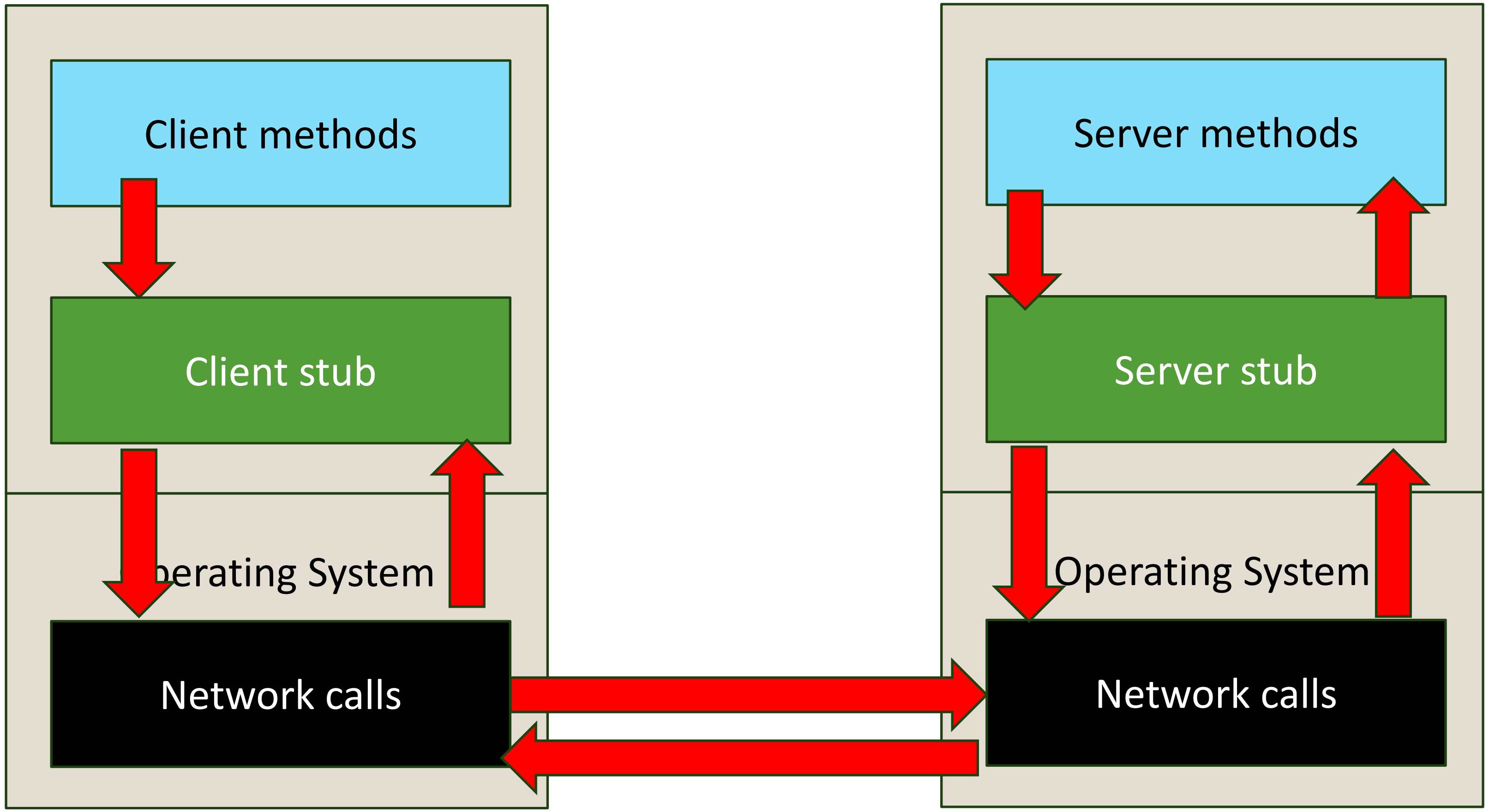
REMOTE PROCEDURE CALLS (RPC)

- 8. Transfer message over the network



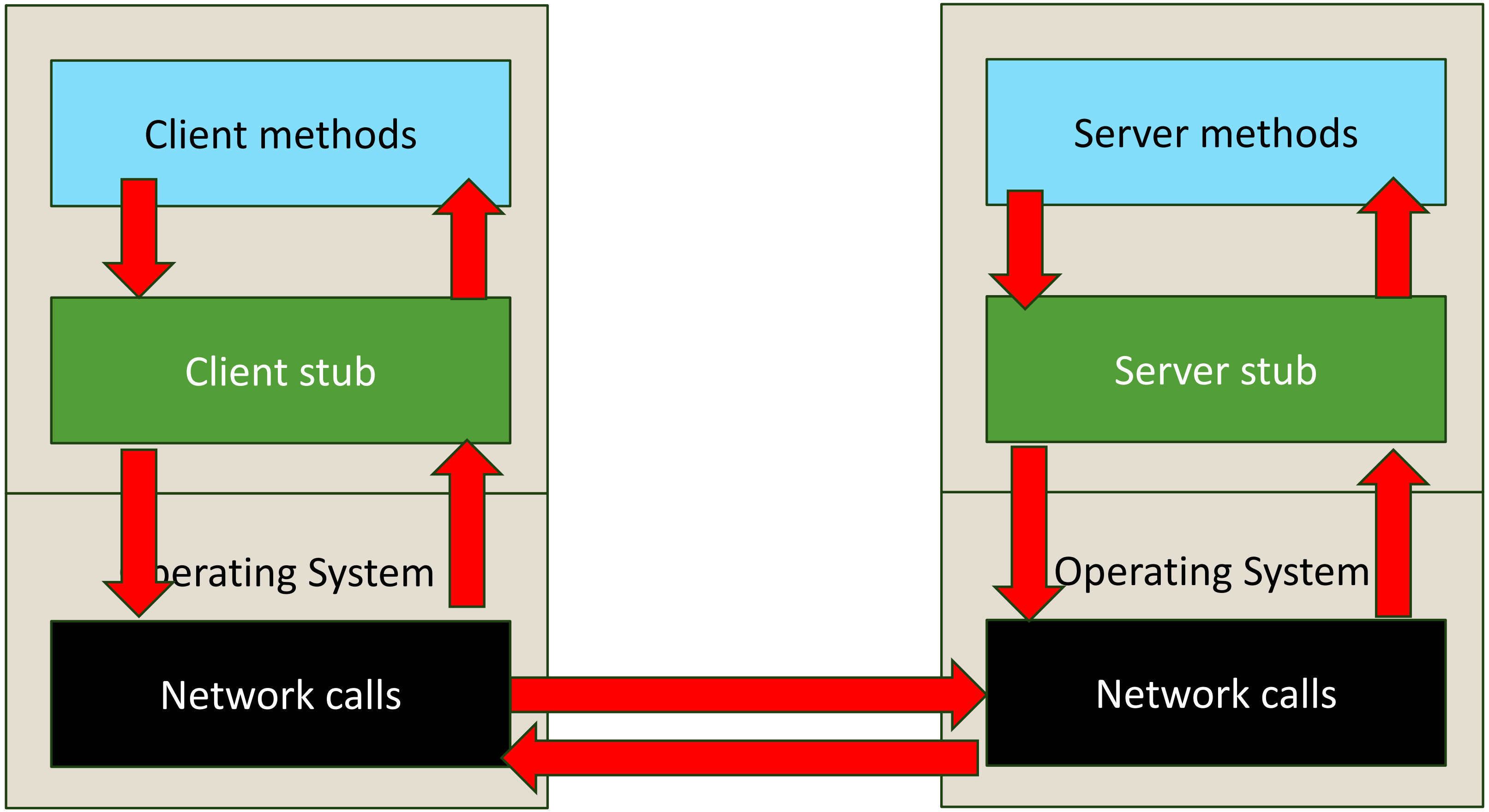
REMOTE PROCEDURE CALLS (RPC)

- 9. Receive message at the client stub



REMOTE PROCEDURE CALLS (RPC)

- 10. unmarshal return values, return to client



REMOTE PROCEDURE CALLS (RPC) – THE GOOD

- Client stub has the same interface as the remote function
- So it looks the same as a local function but:
  - Marshals parameters
  - Sends message
  - Wait for response from server
  - Un-marshal the response and return data
  - Generate exceptions if problems occur

- RPC allows procedure call interface
  - Writing code is simplified
  - No need to worry about sockets, ports, byte ordering etc.

REMOTE PROCEDURE CALLS (RPC) – CHALLANGES

- Transport protocol
  - TCP? UDP? Or HTTP over TCP?
- Error Handling
  - Complicated… prone to errors
- Parameter passing
  - Pass parameters by value (Objects, Data-types) or references/pointers
  - All data must be sent in a pointerless representation
- Service Binding
  - Where/which machine is the server?
  - How do we register server?
  - Need to remember all machines IP addresses and port #s.
  - Remember IP, Ports for each machine (local database?)
- Performance
  - RPC is slower. Why (Compare to local procedure call)
- Security
  - No encryption, so all messages are visible on network
  - Authentication? Client/Server, 3rd party?

PROGRAMMING RPCS

- Language support
  - No default support for RPCs
    - C, C++, Java < 5.0
  - Some support
    - Java > 6, Python, Go etc.
    - No support for heterogeneous environment (e.g. java client talking to python service)
- Solution
  - Interface Definition Language (IDL): Describes RPC procedures
  - Custom Compiler generates client/server stub

ENCODING MESSAGES

RPC: ENCODING MESSAGES

- On local systems, there are no data-type incompatability
  - Int, double, String, Object etc
- Remote machine (incompatabilities occur)
  - Different data type
  - Different size of integer (128-bit, 64-bit, 32-bit, 16-bit etc)
  - Different floating point (IEEE 754, 127-bit, 256-bit, NVIDIA TensorFloar (TF32)
  - Different character set (Unicode, ASCII etc)
  - Different Data Representation

RPC: ENCODING MESSAGES

- Data Representation
  - Big endian: The most significant byte in low memory
    - IP Headers, Java VMs, etc
  - Little endian: The most significant byte in high memory
    - Intel x64, AMD arch.
  - Bi-endian: Processor works with either mode
    - ARM, SPARC V9, IA-64 Intel Itanium

```
byte[] a = new byte[4];
int n = 0x11223344;
a[0] = (byte) n;
a[1] = (byte) (n >> 8);
a[2] = (byte) (n >> 16);
a[3] = (byte) (n >> 24);
System.out.println("%02x, %02x, %02x,
%02x\n", a[0], a[1], a[2], a[3]);
```

Output on an Intel CPU:

44, 33, 22, 11

Output on a PowerPC:

11, 22, 33, 44

RPC: ENCODING MESSAGES

- Serialization
  - Standard encoding technique to enable communication between heterogeneous systems
  - How: Convert data to pointerless format, e.g. array of bytes
  - Examples:
    - JSON (JavaScript Object Notation)
    - XDR (eXternal Data Representation)
    - W3C XML Schema Language
    - ASN.1 (ISO Abstract Syntax Notation)
    - Google Protocol Buffers

RPC: ENCODING MESSAGES

- Serialization
- Two approaches:
  - Implicit type: Send only values; do not send data-types or parameters
    - Ex: ONC XDR
  - Explicit type: Type is sent with each value
    - XML, JSON, ISO ASN.1

RPC: ENCODING MESSAGES

Serialization vs Marshalling

- Serialization: Convert an object to a sequence of bytes that can be transmitted.
- Marshalling: Bundle parameters into a form that can be unmarshalled (re-constructed) by a different process. May include object ID and other state information.
- Marshalling uses serialization

RPC: ENCODING MESSAGES

XML: eXtensible Markup Language

- Benefits:
  - Human read-able
  - Human editable
  - Text structure
- Drawbacks
  - Transmit more data than needed
  - Longer parsing time
  - Data conversion required for numbers

```
<ShoppingCart>
  <items>
    <item>
      <itemID> 1001 </itemID>
      <Title>Iphone 15 Max </Title>
      <Price>5700 </Price>
    </item>
    <item>
      <itemID> 2021 </itemID>
      <Title>Iphone 15 Max Skin </Title>
      <Price>12 </Price>
    </item>
  </items>
</ShoppingCart>
```

RPC: ENCODING MESSAGES

JSON: JavaScript Object Notation

- Light-weight compared to XML
- Based on Javascript
- Human readable
- Explcitly typed
- Includes support for RPC invocation (JSON-RPC)


```
{
  "items": [
    {
      "itemID": 1001,
      "Title": "Iphone 15 Max",
      "Price": "5700.00"
    }
    {
      "itemID": 2021,
      "Title": "Iphone 15 Max Skin",
      "Price": "12.00"
    }
  ]
}
```

RPC: ENCODING MESSAGES

Google Protocol Buffers

- Faster than XML and JSON
- Language Independent
- Each message is a set of names and types
- Used within Google
- 48,000+ message types degined
- Used for RPC and storage

```
message Person{
  required string name = 1;
  required int32 id – 2;
  optional string email = 3;
  enum PhoneType  {
      MOBILE = 0;
      HOME = 1;
      WORK = 2;
  }
  message PhoneNumber
  {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

```
Person person;
person.set_name("John Smith");
person.set_id(1234);
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Learn more about Google Protcol Buffers: https://protobuf.dev/overview/

REMOTE PROCEDURE CALLS (RPC): OPEN NETWORK COMPUTING (ONC) RPC

OPEN NETWORK COMPUTING (ONC)

- Open Network Computing (ONC)
  - A framework for developing distributed computing applications in a network environment.
  - It was initially developed by Sun Microsystems and is commonly associated with the Network File System (NFS) protocol.
  - Provides a set of protocols and APIs (Application Programming Interfaces) enabling communication and resources sharing over a network

OPEN NETWORK COMPUTING (ONC)

- ONC typically includes several key components:
  - **RPC (Remote Procedure Call)**: A protocol that allows a program to execute code on a remote server as if it were local.
  - **NFS (Network File System)**: A protocol that enables remote file systems to be accessed over a network.
  - **XDR (External Data Representation)**: A standard for defining data structures in a platform-independent way, allowing data to be exchanged between systems with different architectures.
- RPC for Unix System V, Linux, BSD, macOS
  - Created by Sun (now Oracle)
  - Defined in RFC 1831 (1995), RFC 5531 (2009)
  - Remains in use mostly because of NFS (Network File System)
- Interfaces defined in an Interface Definition Language (IDL)
  - IDL compiler is rpcgen

OPEN NETWORK COMPUTING (ONC)

rpcgen name.x

- produces:
  - name.h header
  - name_svc.c server skeleton (stub)
  - name_clnt.c client stub (proxy)
  - [name_xdr.c] optional XDR data conversion routines
- Function names derived from IDL function names and version numbers
- Client gets **pointer** to result
  - Allows it to identify failed RPC (null return)
  - Reminder: C doesn't have exceptions!

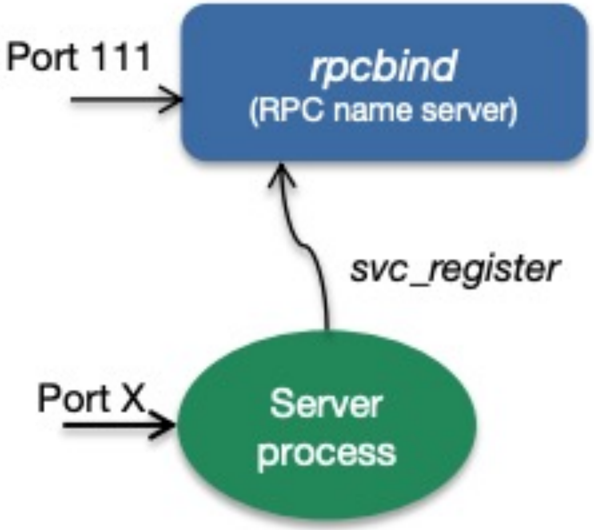
```
name.x

program GETNAME {
  version GET_VERS {
    long GET_ID(string) = 1;
    string GET_ADDR(long) = 2;
  } = 1; /* version */
  version GET_VERS2 {
    long GET_ID(string) = 1;
    string GET_ADDR(string) = 2;
  } = 2; /* version */
} = 0x31223456;
```

Interface definition: version 2

OPEN NETWORK COMPUTING (ONC)

Server

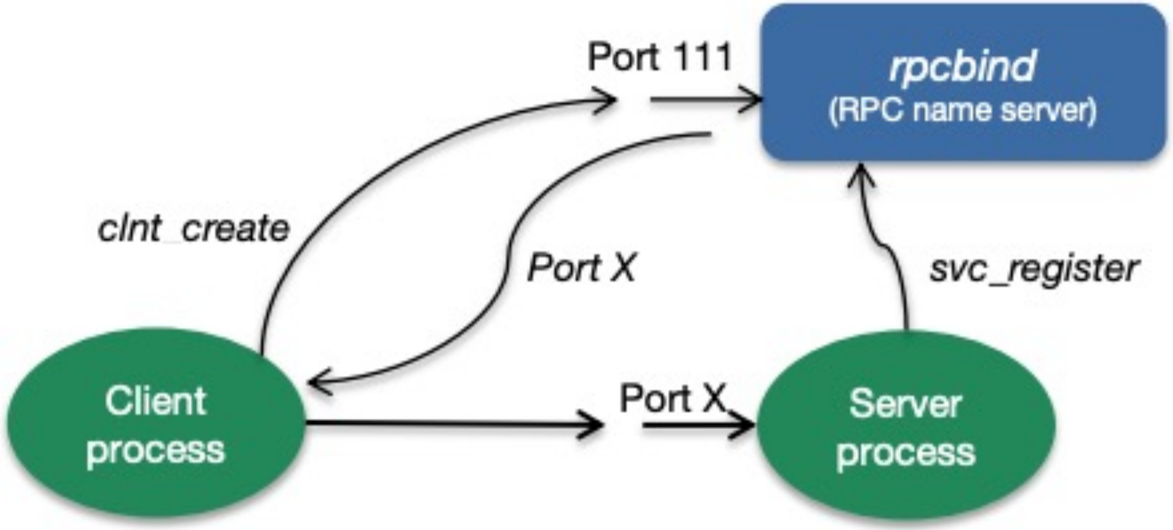
- Creates a socket, binds to "any" available local port
- Calls a function in the RPC library:
  - **Svc_register**, register program #, port #, protocol (TCP/UDP)
  - Contacts port_mapper, **rpcbind**
    - Name server
    - Keep track of {program#, version#, protocol -> port# etc}
- Server then listens and waits to accept client connections



OPEN NETWORK COMPUTING (ONC)

Client

- Calls **clnt_create** {Server_Name, program#, Version#, Protocol (TCP/UDP)}
- **Clnt_create** contact port mapper on the server to bind port (done once)
- Communications:
  - Marshalling to XDR format (eXternal Data Representation)



OPEN NETWORK COMPUTING (ONC)

Whats good!

- No need to worry about unique port for binding
- Protocol can be selected at run-time
- Programmer: No need to worry about message boundaries, fragmentation, disassembly/re-assembly.
- Application: Need to know only ONE transport address (rpcbind process)
- Function call instead of send/receive
- Versioning support between client & server

Challenges

- Managing multiple machines (Need to know which machine provides service)
- Distributed Computing Environment (DCE) RPC improved Sun RPC

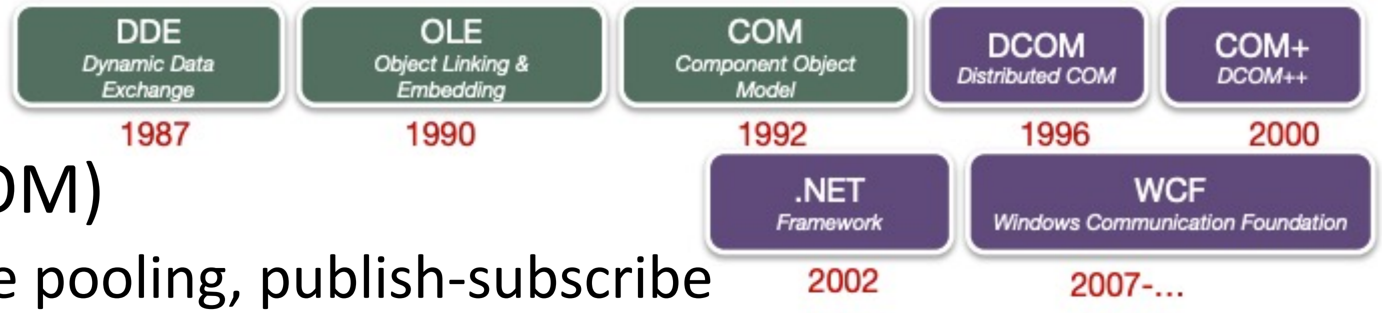
OPEN NETWORK COMPUTING (ONC)

Distributed Computing Environment (DCE) RPC

- Improved Sun RPC
  - DCE RPC uses Interface Definition Language (IDL) to define the interfaces and data structures
  - DCE RPC includes built-in support for security features such as authentication, encryption, and access control
  - DCE RPC provides mechanisms for error detection and handling
- Superseded by RESTful APIs

REMOTE PROCEDURE CALLS (RPC): MICROSOFT TECHNOLOGIES

MICROSOFT COM+/DCOM



- COM+: Windows 2000
- Component Object Model (COM)
  - Supports transactions, resource pooling, publish-subscribe communications
- Service Control Manager (SCM)
  - Starts at OS boot.
  - Works as a RPC server
  - Maintains a Database of installed devices
  - Requests creation of object on server
- Surrogate process runs components: **dllhost.exe**
  - A process that loads DLL-based COM objects
- Multi-threaded: Can handle multiple clients simultaneously

MICROSOFT COM+/DCOM

- Communication through ObjectRPC (ORPC)
  - Based on DCE RPC protocol
- Marshalling mechanism: NDR
  - Same as Network Data Representation used by DCE RPC
- Microsoft Interface Definition Language (MIDL)
  - MIDL files are compiled with a IDL compiler
  - Same as DCE IDN
- Generates C++ code for marshalling, unmarshalling & stubs

– Client side is called the *proxy*
– Server side is called the *stub*

Both are COM objects that are loaded by the COM libraries as needed: the application loads the client COM object, which contacts the server to load the server COM object

MICROSOFT COM+/DCOM

- Microsoft Contributions
  - Object Lifetime (terminate after time expired)
  - Abnormal Client termination (terminate non-responding clients)
  - Client Pinging (Heart-beat / Breathing – ensure the clients are "awake")
  - Fits into Microsoft COM model
  - Generic server hosts dynamically loaded objects
    - Deal with "dead" clients
    - Heart-beat counting and pinging
  - Works only with Microsoft technologies!

REMOTE PROCEDURE CALLS (RPC): JAVA RMI

JAVA RMI

Java RMI (Remote Method Invocation) is a Java API

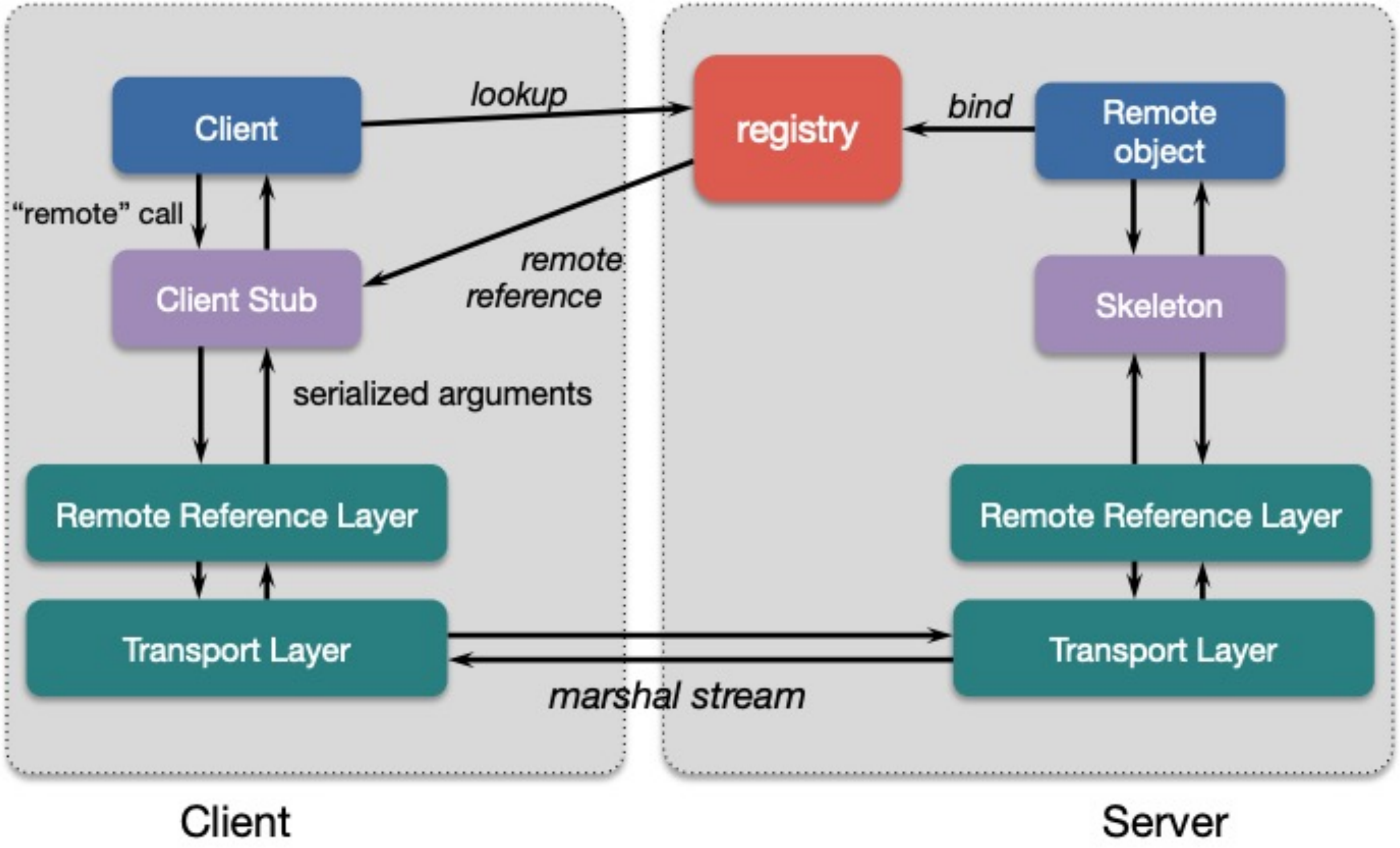
- Facilitates communication between different Java Virtual Machines (JVMs) over a network
- Allows Java objects to invoke methods on remote Java objects residing in different JVMs

RMI is built for Java only!

- No goal of OS interoperability
- No language interoperability
- No architecture interoperability
- No need for external data representation
- All sides run a JVM
- Benefit: simple and clean design

JAVA RMI

- Client: Invokes method on remote object
- Server: Process that owns the remote object
- Registry: Nameserver that relates objects with names



– Skeleton

Server-side code that calls the actual remote object implementation

– Stub

Client-side proxy for the remote object

Communicates method invocations on remote objects to the server

JAVA RMI

1. Interface Definition: Define interfaces that describe the methods that will be invoked remotely.

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyRemoteInterface extends Remote {
    // Remote method declaration
    public String sayHello() throws RemoteException;
}
```

JAVA RMI

2. Implementation: Provide an implementation of the remote interface. This implementation class must extend **java.rmi.server.UnicastRemoteObject** and implement the remote interface.

```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MyRemoteObject extends UnicastRemoteObject
        implements MyRemoteInterface {
    public MyRemoteObject() throws RemoteException {
        super();
    }

    // Implementation of the remote method
    public String sayHello() throws RemoteException {
        return "Hello from server!";
    }
}
```

JAVA RMI

3. Server Setup: Create and start an RMI registry on the server side. The ***RMI registry*** provides a naming service that allows clients to look up remote objects by name. Here we “bind” “MyRemoteObject” to the registry.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Server {
    public static void main(String[] args) throws Exception {
        // Create and export the remote object
        MyRemoteInterface remoteObject = new MyRemoteObject();
        // Bind the remote object to the RMI registry
        Registry registry = LocateRegistry.createRegistry(1099);
        registry.rebind("MyRemoteObject", remoteObject);
        System.out.println("Server ready");
    }
}
```

JAVA RMI

4. Client Invocation: On the client side, look up the remote object from the ***RMI registry*** using its name and then invoke its methods as if they were local. Here **LocateRegistry.getRegistry** binds to the registry; the **registry.lookup** finds the "MyRemoteObject" object.

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {
    public static void main(String[] args) throws Exception {
        // Look up the remote object from the RMI registry
        Registry registry = LocateRegistry.getRegistry("localhost", 1099);
        MyRemoteInterface remoteObject =
                (MyRemoteInterface) registry.lookup("MyRemoteObject");

        // Invoke remote method
        String result = remoteObject.sayHello();
        System.out.println("Result from server: " + result);
    }
}
```

JAVA RMI

Similarity to local objects

- References to remote objects can be passed as parameters
- You can execute methods on a remote object
- Objects can be passed as parameters to remote methods
- Object can be cast to any of the set of interfaces supported by the implementation
- Operations can be invoked on these objects

JAVA RMI

Differences:

- Objects (parameters or return data) passed by value
  - Changes will visible only locally
- Remote objects are passed by reference
  - Not by copying remote implementation
  - The “reference” is not a pointer. It’s a data structure: { IP address, port, time, object #, interface of remote object }
- RMI generates extra exceptions

REMOTE PROCEDURE CALLS (RPC): PYTHON RPYC

RPC IN PYTHON

- Various implementations of RPC in Python:
  - xmlRPC, PyRO, PyInvoke, RPyC, ZeroRPC
- General idea of implementing RPC on Python
  - Create a connection using an RPC object
  - Then invoke remote methods using that object

RPC IN PYTHON

Example using xmlrpc

1. Define the Server Method: Define the method that will be remotely accessible on the server side.

```
def add(x, y):
    return x + y
```

RPC IN PYTHON

2. Expose the Method with XML-RPC: Use the SimpleXMLRPCServer class from the xmlrpc.server module to create an XML-RPC server. Register the method using the register_function method.

```
from xmlrpc.server import SimpleXMLRPCServer

server = SimpleXMLRPCServer(('localhost', 8000))
server.register_function(add, 'add')
```

RPC IN PYTHON

3. Start the Server:

```
server.serve_forever()
```

RPC IN PYTHON

4. Invoke the Remote Method: On the client side, use the xmlrpc.client module to create an XML-RPC proxy object that connects to the server. Then, call the remote method through this proxy object.

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy('http://localhost:8000/')
result = proxy.add(3, 5)
print(result)  # Output: 8
```

RPC IN PYTHON

Example: Server

```
from xmlrpc.server import SimpleXMLRPCServer

# Define a function to be exposed remotely
def add(x, y):
    return x + y

# Create an XML-RPC server
server = SimpleXMLRPCServer(('localhost', 8000))
server.register_function(add, 'add')

print("Server listening on port 8000...")
# Start the server
server.serve_forever()
```

RPC IN PYTHON

Example: Client

```python
import xmlrpc.client

# Create an XML-RPC proxy object
proxy = xmlrpc.client.ServerProxy('http://localhost:8000/')

# Call the remote method through the proxy object
result = proxy.add(3, 5)
print("Result from server:", result)
# Output: Result from server: 8
```

RPC IN PYTHON

Example using Remote Python Call (RPyC) Library

- Define a Service MyService.
- Add remote method exposed_add to the service
- Start the server

```
import rpyc

# Define a service class
class MyService(rpyc.Service):
    def exposed_add(self, x, y):
        return x + y

# Start the RPyC server
if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    server = ThreadedServer(MyService, port=5000)
    print("Server started on port 5000...")
    server.start()
```

RPC IN PYTHON

- Connect to the server
- Call the add method on the root object, which is a proxy for the MyService instance running on the server.

```
import rpyc

# Connect to the RPyC server
conn = rpyc.connect("localhost", 5000)

# Call the remote method
result = conn.root.add(3, 5)
print("Result from server:", result)
# Output: Result from server: 8
```

RPC IN PYTHON

- Transparent RPC interface
  - No definition files, stub compilers, name servers, transport services
- Symmetric operation
  - Both sides can invoke RPCs on each other; enables callback functions
- Server
  - RPyC ThreadedServer started on the server program
  - Binds to a default port (18812) or you specify the host's IP address and port
- Client
  - Connects to the server
  - Performs remote operations through the modules property, which exposes the server module's namespace

REMOTE PROCEDURE CALLS (RPC): IN A NUTSHELL

RPC IN A NUTSHELL

- **Marshalling operations**:
  - Serialization and deserialization of data for transmission over the network.
  - Addition of metadata such as function/method calls, object instances, and version numbers.
  - Common serialization formats include XML (for XML-RPC), JSON (for JSON-RPC), and Protocol Buffers (for gRPC).
- **Name service and discovery operations**:
  - Registration and lookup of binding information including ports, machines, and protocols.
  - Support for dynamic port assignment by the operating system.

RPC IN A NUTSHELL

- **Transport protocol support**:
  - Utilization of transport protocols like TCP, UDP, or HTTP/HTTPS (for XML-RPC).
  - gRPC employs HTTP/2 over TCP for data transmission.
- **Connection Management**:
  - Handling creation, maintenance, and termination of network connections.
  - Addressing concerns such as connection pooling, retries, timeouts, etc.

RPC IN A NUTSHELL

- **Service definition and stub/skeleton generation**:
  - Explicit definition of service interfaces using interface definition languages.
  - Automatic generation of stubs (client-side proxies) and skeletons (server-side method implementations) from these definitions.
- **Security operations**:
  - Authentication and authorization mechanisms for client-server authentication and secure communication channels.
  - Encryption techniques like TLS for data security.

RPC IN A NUTSHELL

- **Stub memory management and garbage collection**:
  - Memory allocation and deallocation for storing parameters and network buffers.
  - Tracking object references and managing memory for object deletion.
- **Error Handling**:
  - Robust error handling for application and network-level errors during remote calls.
  - Support for exception propagation.
- **Object and function ID operations**:
  - Support for passing references to remote functions or objects across processes (not universally supported).