Remote Procedure Calls and An Implementation Example

CS587x Lecture 5
Department of Computer Science
Iowa State University

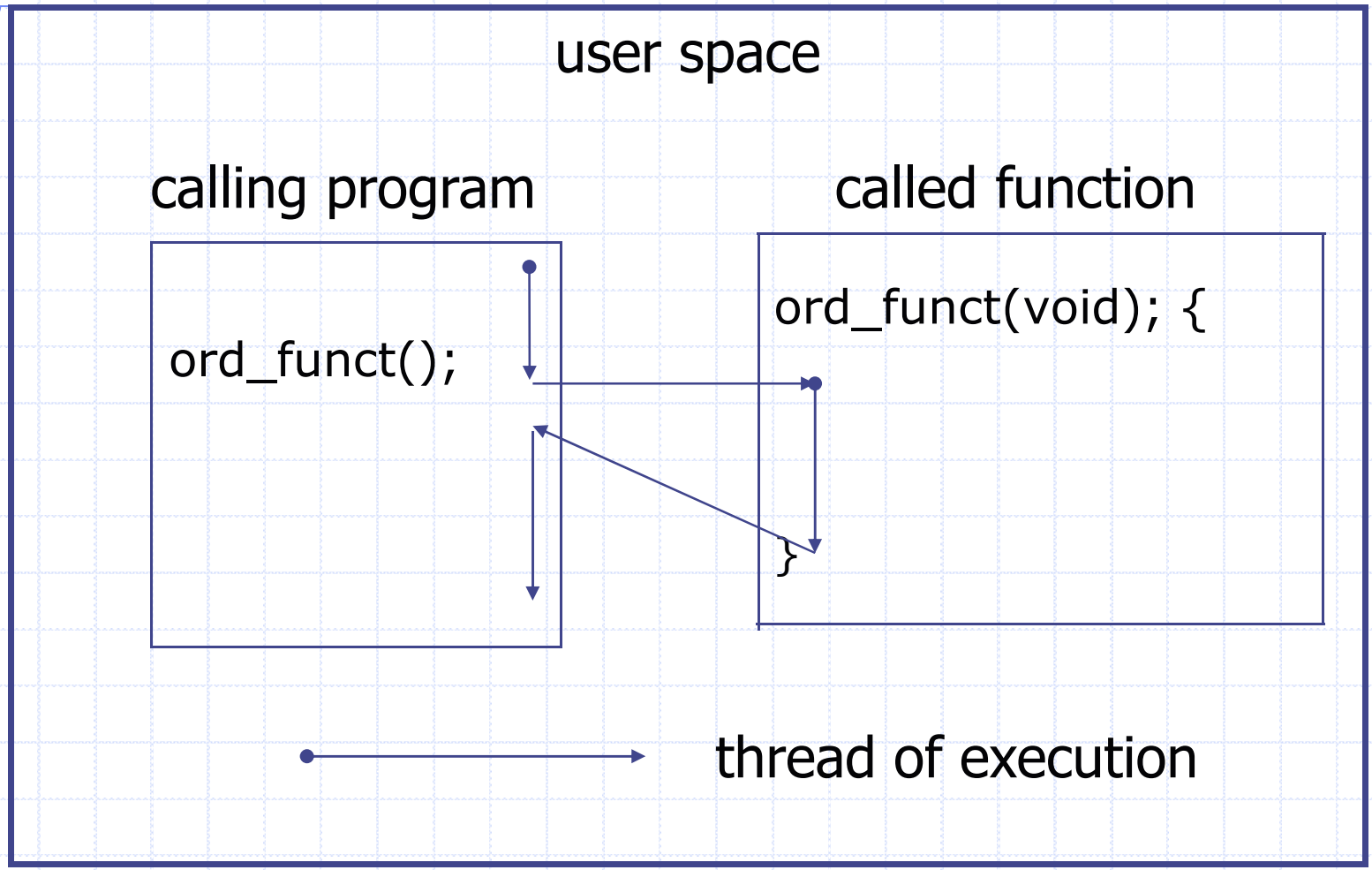
What to cover today

- Concept of RPC
- An implementation example
  - Java_to_C (J2C) RPC

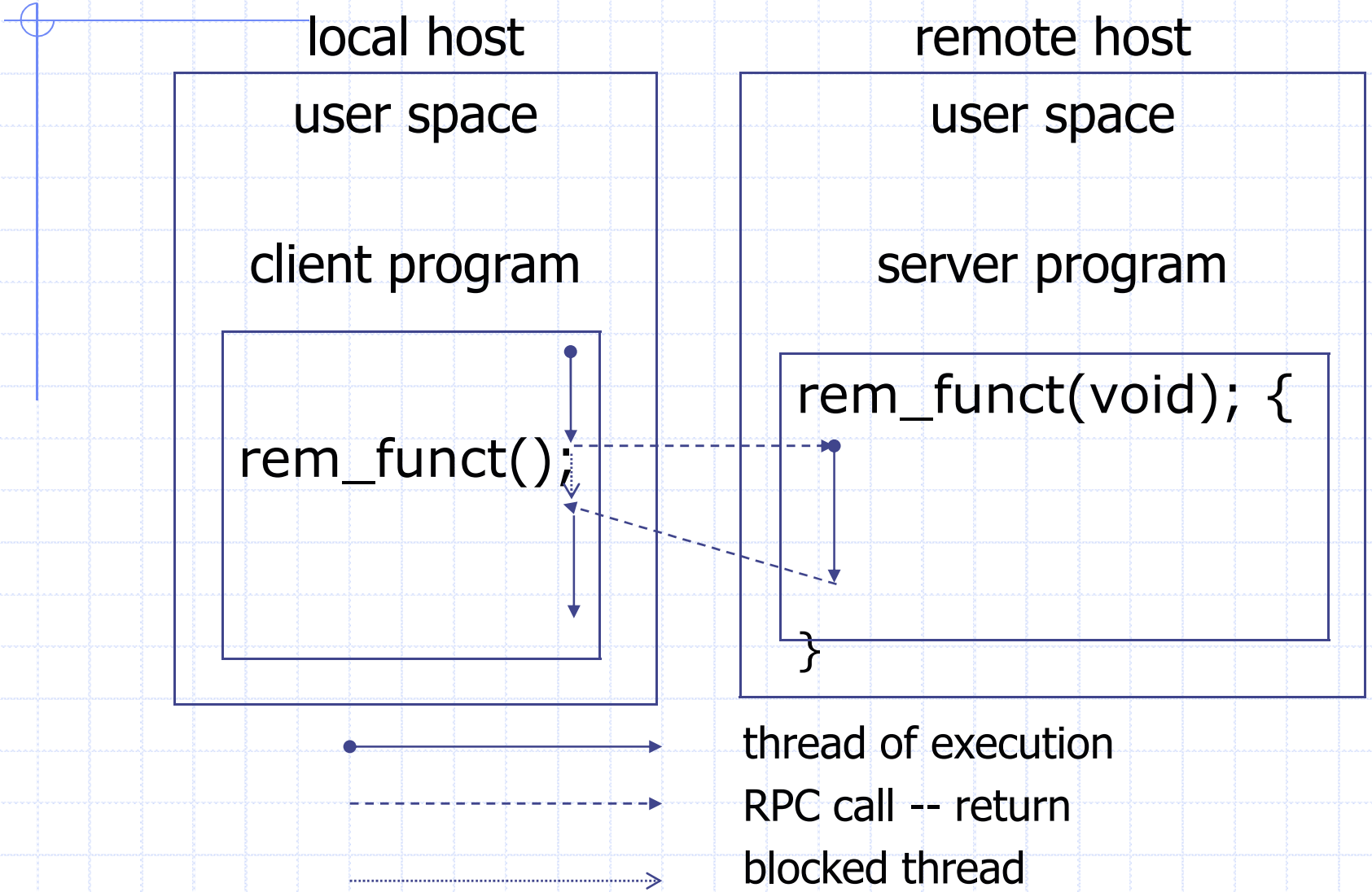
Remote Procedure Call

- What is RPC for?
  - Allowing programs to call procedures located on other machine transparently
  - Send/Receive do not conceal communication
- Scope of use
  - Distributed computing
    - Task and data partitioned environments
    - Task distribution
      - Front-end load-balances across functional back ends
  - Services
    - Client-server model
    - Mail servers, databases (transaction servers)

Ordinary Function Call



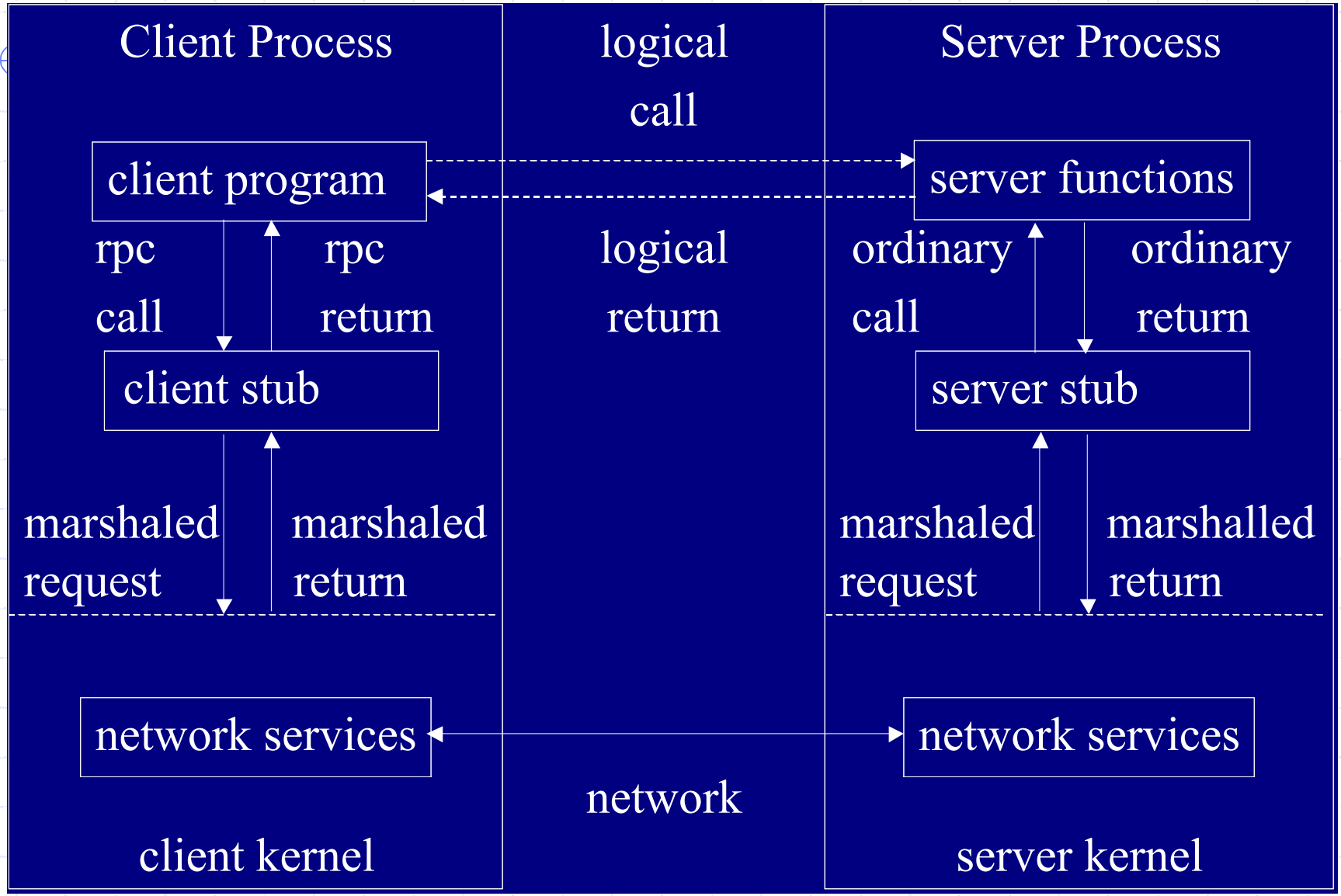
Remote Procedure Call



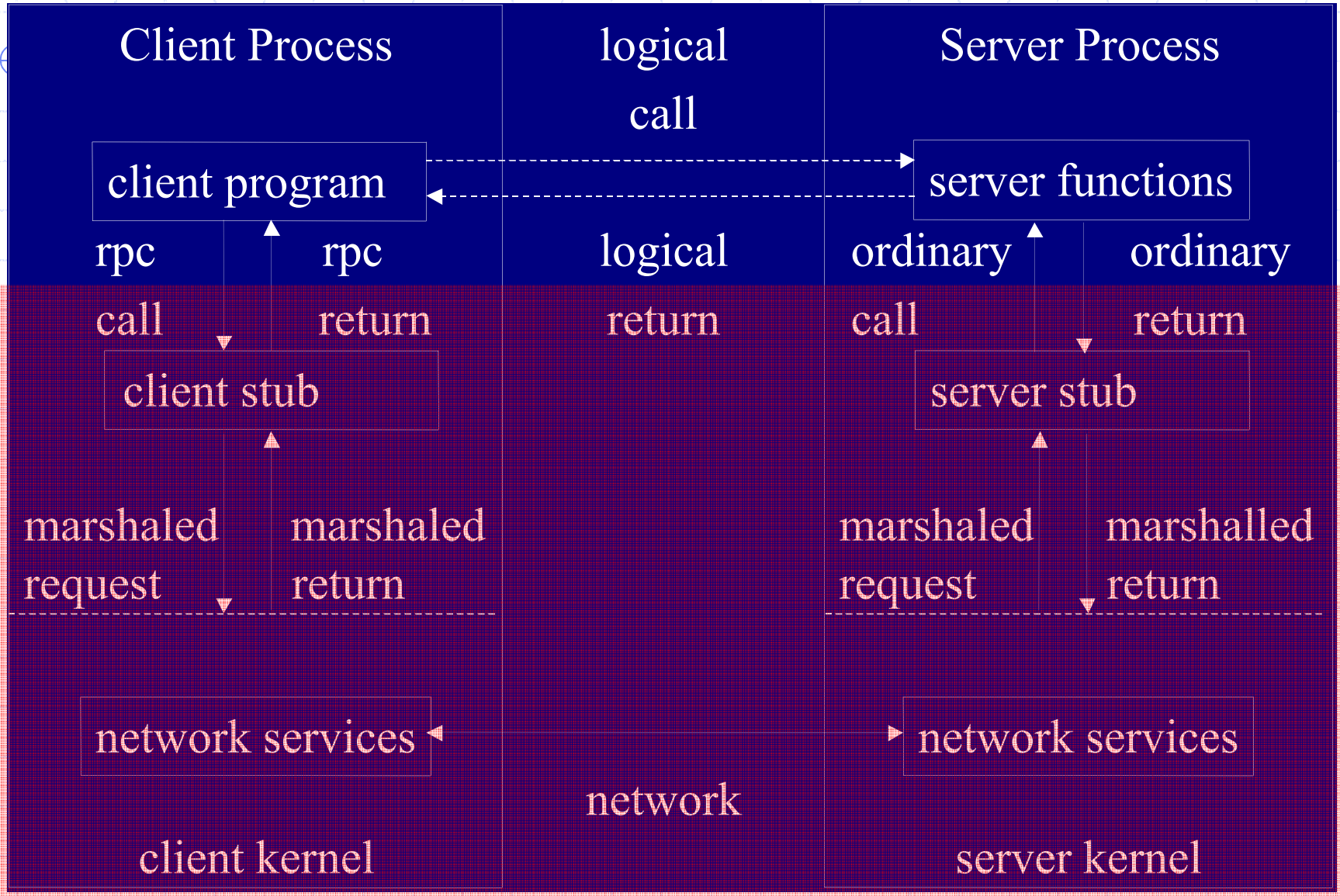
RPC Goals

- Client program only sees an ordinary function call to the client stub
- Server functions are ordinary functions
- The underlying mechanism for transporting requests and returning them should be transparent to the programmer
- RPC should be independent of the transport protocol

How RPC Works?



How RPC Works?



Client Stub

- Responsible for
  - Converting arguments and assembling them into a message for network transmission
  - Sending the message to the specified remote machine and receiving the response back
  - Passing the response to the caller

Marshaling

- Conversion to a network message is called marshaling the arguments
- Converts to machine independent format so machines with different architectures can participate (e.g., XDR - external data representation)
- Then the client stub makes a system call to the kernel of the OS (e.g., using TCP/UDP sockets) to send the message over the network and the client stub waits for a reply

Server Stub

- When a client request arrives, the server kernel passes it to the waiting server stub
- The server stub unmarshals the arguments and calls the requested service as a local function call
- When the function call returns, the server stub marshals the return values into an appropriate network message and performs a system call (e.g., using TCP/UDP sockets) to transmit the message to the client

RPC Programming Steps

- Define remote APIs using IDL (Interface Definition Language)
  - Specify function parameters (i.e., types)
- Run IDL compiler to generate server and client stubs
- Implement the remote APIs
- Compile the entire set of programs
  - your own code
  - the stub files generated by IDL compiler

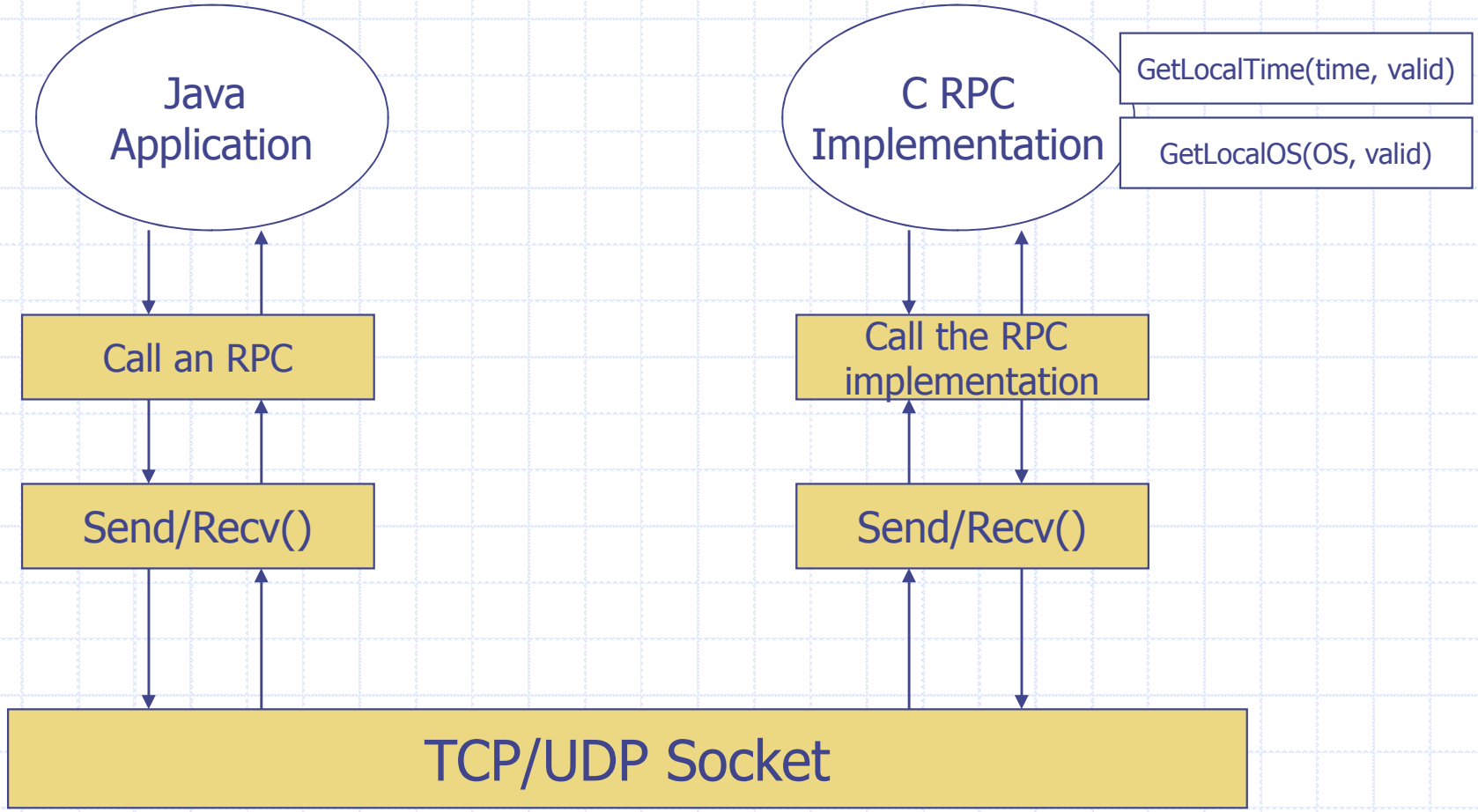
Implementation Issues

- How to marshal/unmarshal messages?
  - Data type conversion
  - Big/little-endian conversion
  - Parameter passing
    - Reference/value (c)
    - Object serialization (java)

Java-to-C (J2C) RPC

- A hard implementation
  - Not flexible
  - Hard to maintain/upgrade
- A general implementation of J2C
  - Supported data types
  - Generic marshalling/unmarshalling
  - J2C compiler
    - Given a C function, automatically generates its corresponding Java class
    - Make RPC communication transparent

Java-to-C (J2C) RPC



Interface: How to Make an RPC?

- C Interface
  - How to call its C implementation?
- Java Interface
  - How to represent a C function in Java
  - How to set inputs
  - How to execute
  - How to get outputs

Example: GetLocalTime()

```
typedef struct
{
      int     *time;
      char    *valid;
} GET_LOCAL_TIME;

void GetLocalTime(GET_LOCAL_TIME *ds);
```

C Interface Design

- Call standard function implementation
  - e.g., GetLocalTime(char *buffer)

Java Interface Design

- Each RPC is associated with a class

```
class GetLocaltime();
```

- Steps of making an RPC call

1. Instantiate an RPC object

```
obj = new GetLocalTime();
```

2. Set inputs

```
obj.valid.setValue(FALSE);
```

3. Execute

```
obj.execute(IP, PORT);
```

4. Get outputs

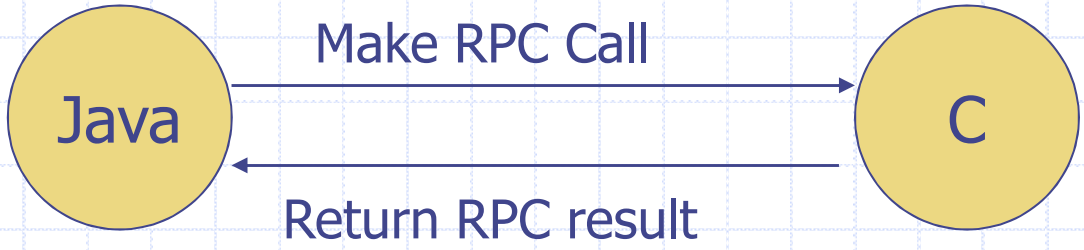
```
int t = obj.time.getValue();
```

RPC Class of GetLocalTime()

```
class GetLocalTime
{
    c_int    time;
    c_char   valid;

    public int execute(string IP, int port);
}

class c_int
{
    byte[] buf = byte[4];

    public int getSize();
    public int getValue();
    public void setValue(byte[] buf);
    public void setValue(int v);
    public byte[] toByte();
}
```

Implementation of execute()

- Communication protocol



Length (4 bytes): the length of CmdID+CmdBuf
CmdID (100 bytes): the command ID
CmdBuf (dynamic): the parameters to the command

Implementation of Execute()

- Create a binary buffer

```
1. int length = 100+time.getsize()+valid.getsize();
2. byte[] buf = new byte[4+length];
```

- Marshall parameters into the buffer

```
1. buf[0, 4] = length; offset = 4;
2. buf[offset, 100] = "GetLocalTime"; offset^^
3. buf[offset, time.getSize()] = time.toByte(); offset^^
4. buf[offset, valid.getSize()] = valid.toByte();
```

- Send/receive the buffer to/from the RPC server

```
1. s = CreateSocket(IP, port);
2. SendPacket(s, buf, buf.length());
3. RecvPacket(s, buf, buf.length());
```

- Set parameters according to the buffer

```
1. time.setValue(buf, 100);
2. valid.setValue(buf, 100+time.getSize());
```

C Implementation

- Receive a command

```
1.  s = CreateSocket(port);
2.  length = new byte[4];
3.  RecvPacket(length, 4);
4.  buf = new byte[length];
5.  RecvPacket(s, buf, length);
```

- Execute the command

```
1.  switch buf[0-99] of
2.      case "GetLocalTime":
3.      {
4.          ds = malloc(sizeof(GET_LOCAL_TIME));
5.          ds.time=&buf[100];
6.          ds.valid = &buf[100, sizeof(time field)];
7.          GetLocalTime(&ds);
8.          free(ds);
9.          break;
6.      }
```

- Send the command back

```
1.  SendPacket(s, buf, length);
```

Problems of Hard Implementation

- A new command needs to be added?
- An existing command needs to be deleted?
- Some parameters to a command need to be changed?
  - Add a new field
  - Delete an existing field
  - Change the type of an existing field

A General Implementation

- Supported data types
- Generic marshalling/unmarshalling
- J2C compiler
  - Given a C function, automatically generates its corresponding Java class
  - Make RPC communication transparent

RPC Class of GetLocalTime()

```
class GetLocalTime
{
    c_int    time;
    c_char   valid;

    public int execute(string IP, int port);
}

class c_int
{
    byte[] buf = byte[4];

    public int getSize();
    public int getValue();
    public void setValue(byte[] buf);
    public void setValue(int v);
    public byte[] toByte();
}
```

Implementation of Execute()

- Create a binary buffer

```
1. int length = 100+time.getsize()+valid.getsize();
2. byte[] buf = new byte[4+length];
```

- Marshall parameters into the buffer

```
1. buf[0, 4] = length; offset = 4;
2. buf[offset, 100] = "GetLocalTime"; offset^^
3. buf[offset, time.getSize()] = time.toByte(); offset^^
4. buf[offset, valid.getSize()] = valid.toByte();
```

- Send/receive the buffer to/from the RPC server

```
1. s = CreateSocket(IP, port);
2. SendPacket(s, buf, buf.length());
3. RecvPacket(s, buf, buf.length());
```

- Set parameters according to the buffer

```
1. time.setValue(buf, 100);
2. valid.setValue(buf, 100+time.getSize());
```

C Implementation

- Receive a command

```
1.  s = CreateSocket(port);
2.  length = new byte[4];
3.  RecvPacket(length, 4);
4.  buf = new byte[length];
5.  RecvPacket(s, buf, length);
```

- Execute the command

```
1.  switch buf[0-99] of
2.      case "GetLocalTime":
3.      {
4.          ds = malloc(sizeof(GET_LOCAL_TIME));
5.          ds.time=&buf[100];
6.          ds.valid = &buf[100, sizeof(time field)];
7.          GetLocalTime(&ds);
8.          free(ds);
9.          break;
6.      }
```

- Send the command back

```
1.  SendPacket(s, buf, length);
```

Can a Tool Does This?

- Given an RPC definition (i.e., a C data structure), the tool should
  - generate the corresponding RPC class
  - make the communication of RPC transparent to users, i.e., when call Execute(),
    - marshal the parameters
    - send/recv command to/from the RPC server
    - set the parameters accordingly

Challenge –

Given an RPC definition, can we make a tool to generate the red codes accordingly?

Keys to the Solution

- Define a generic RPC model
  - generic data structure and field
- RPC Implementation replies only on the generic model
  - Parameter marshalling
  - Execution
  - Parameter unmarshalling
- Based on an RPC definition, we need to generate only its corresponding RPC class

What Defines a Data Structure

- struct = name + a list of fields
- What can be changed?
  - Name of data structure (i.e., RPC)
  - Number of fields
  - Each field
    - Data type
    - Variable name

```
typedef struct
{
     int     *time;
     char    *valid;
} GET_LOCAL_TIME;
```

What Defines a Field

- Field = type + name
- Primitive data type
  - int (4 bytes)
  - short (2 bytes)
  - char (1 bytes)
  - etc.
- Complex data type
  - data structure
  - array

```
typedef struct
{
    int     x;
    char    y;
    short   z[20];
} DS1;

typedef struct
{
    DS1     x1[100];
    DS2    *x2;
} DS2;
```

Data Structure Abstraction

```
public abstract class BaseStruct
{
    String      Name;
    BaseField   Field[] = null;

    public byte[] toByte()
    {
        for (int i=0; i<Field.length; i++)
        {
           buf = buf + Field[i].toByte();
        }
    }

    public void setValue(byte[] buf} {…}
    public int getSize(){…};
}
```

Field Abstraction

```
public abstract class BaseField
{
    String          Name;

    BaseType        BType           = null;
    BaseType        BTypeArray[]    = null;
    BaseStruct      BStruct         = null;
    BaseStruct      BStructArray[] = null;

    public BaseField(String name, BaseType bt)
    { Name = name; Btype = bt }
    public BaseField(String name, BaseType bta[]){…}
    public BaseField(String name, BaseStruct bs){…}
    public BaseField(String name, BaseStruct bsa[]){…}

    public byte[] toByte();
    public byte[] setvalue(byte buf[]);
    public int getSize();
}
```

Primitive Type Abstraction

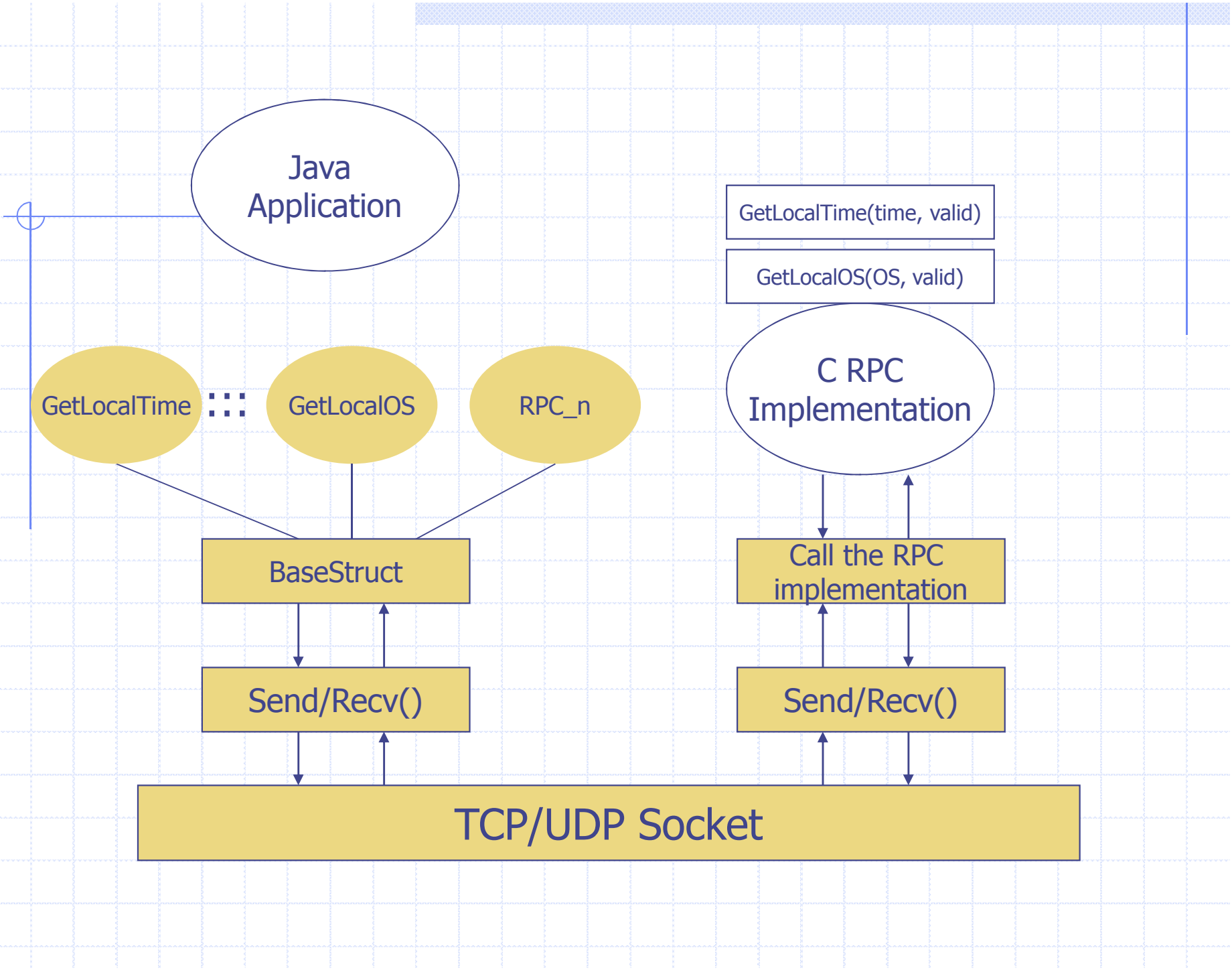
```
public abstract class BaseType
{
    byte buffer[];
    int  myType;

    public byte[] toByte();
    public byte[] setvalue(byte buf[]);
    public getSize();
}
```

```
public class U8 extends BaseType
{
    public U8(char value)
    {
       buffer = new byte[1];
       buffer[0] = value;
       myType = TYPE_U8;
    }
}
```

Primitive Array Abstraction

```
public class BaseArray extends BaseType
{
    int ArrayType;
    public BaseArray(int type, int array_size);
    public int getSize();
}
```

```
public class U8_ARRAY extends BaseArray
{
    public U8_ARRAY(int size)
    {
       super(TYPE_U8_ARRAY, size);
    }
}
```


Java Application
GetLocalTime(time, valid)
GetLocalOS(OS, valid)
C RPC Implementation
GetLocalTime
GetLocalOS
RPC_n
BaseStruct
Call the RPC implementation
Send/Recv()
Send/Recv()
TCP/UDP Socket

Implementation of DS.Execute()

- Create a binary buffer

```
int length = 100;
for (int i=0; i<ds.getFieldNumber(); i++)
{
    length = length + ds.field[i].getsize();
}
byte[] buf = new byte[4+length];
```

- Marshall parameters into the buffer

```
buf[0, 4] = length; offset = 4;
buf[offset, 100] = ds.getName(); offset = offset + 100;
for (int i=0; i<ds.getFieldNumber(); i++)
{
    buf[offset, ds.filed[i].getSize()] = ds.field[i].toByte();
    offset = offset + ds.filed[i].getSize();
}
```

- Send/receive the buffer to/from the RPC server

```
s = CreateSocket(IP, port);
SendPacket(s, buf, buf.length());
RecvPacket(s, buf, buf.length());
```

- Set parameters according to the buffer

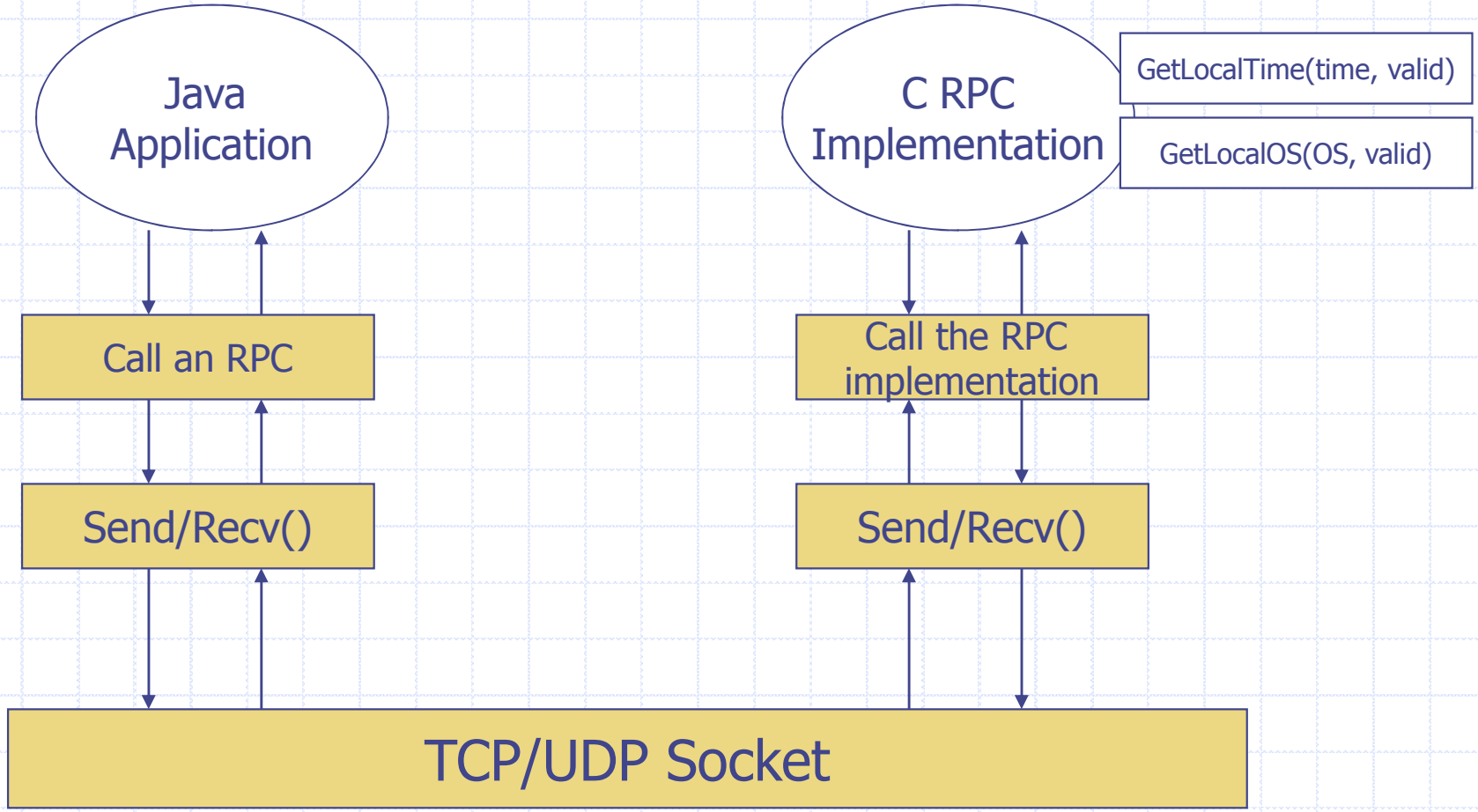
```
offset = 100;
for (int i=0; i<ds.getFieldNumber(); i++)
{
    Ds.field[i].setValue(buf, offset);
    offset = offset + ds.field[i].getSize();
}
```

Equavalent RPC Class

```
typedef struct
{
    int     x;
    char    y;
    short   z[20];
} DS1;

typedef struct
{
    DS1     x1[100];
    DS2    *x2;
} DS2;
```

```
public class DS1
{
    S32 x = new S32();
    U8  y = new U8();
    S16 z[] = new S16[20];
}

public class DS2
{
    DS1 x1[] = new DS1[100];
    DS2 x2 = new DS2();
} DS2;
```

Testing J2C


Java Application
C RPC Implementation
GetLocalTime(time, valid)
GetLocalOS(OS, valid)
Call an RPC
Call the RPC implementation
Send/Recv()
Send/Recv()
TCP/UDP Socket