Remote Procedure Calls and An Implementation Example

Concept of RPC
An implementation example
Java_to_C (J2C) RPC

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

Calling program
ord_funct();
ord_funct(void); {
}

Called function

Thread of execution
Remote Procedure Call

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?

Client Process
- client program
- rpc call
- client stub
- marshaled request
- network services
- client kernel

logical call
marshaled return

Server Process
- server functions
- ordinary call
- server stub
- marshaled request
- network services
- server kernel

logical return

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
**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

- 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[] toBytes();
}
Implementation of execute()

Communication protocol

Java
Make RPC Call
C

Return RPC result

Length | CmdID | CmdBuf
---|---|---
Length (4 bytes): the length of CmdID+CmBuf
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 = 4;
3. buf[offset, time.getSize()] = time.toString(); offset = 4;
4. buf[offset, valid.getSize()] = valid.toString();

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());

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()

```java
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()

1. Create a binary buffer
   - int length = 100 + time.getSize() + valid.getSize();
   - byte[] buf = new byte[4 + length];
2. Marshall parameters into the buffer
   - buf[0, 4] = length; offset = 4;
   - buf[offset, 100] = "GetLocalTime"; offset += 100;
   - buf[offset, time.getSize()] = time.toByte(); offset += time.getSize();
   - buf[offset, valid.getSize()] = valid.toByte(); offset += valid.getSize();
3. Send/receive the buffer to/from the RPC server
   - s = CreateSocket(IP, port);
   - SendPacket(s, buf, buf.length());
   - RecvPacket(s, buf, buf.length());
4. Set parameters according to the buffer
   - time.setValue(buf, 100);
   - valid.setValue(buf, 100 + time.getSize());

C Implementation

1. Receive a command
   - s = CreateSocket(port);
   - length = new byte[4];
   - RecvPacket(s, buf, length);
   - buf = new byte[length];
   - RecvPacket(s, buf, buf.length());
2. Execute the command
   - switch buf[0-99] of
     - case "GetLocalTime":
       - (ds = malloc(sizeof(GET_LOCAL_TIME)));
       - ds.time = &buf[100];
       - ds.valid = &buf[100 + time.getSize()];
       - GetLocalTime(&ds);
       - free(ds);
       - break;
     - 6.
   - 7.
   - 8.
   - 9.
   - 10.
3. Send the command back
   - 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

```c
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**

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

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

**Java Application**

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

**C RPC Implementation**

```
GetLocalTime(time, valid)
GetLocalOS(OS, valid)
BaseStruct
Send/Recv()
TCP/UDP Socket
Send/Recv()
```

**Implementation of DS.Execute()**

- Create a binary buffer
  ```java
  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
  ```java
  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.field[i].getSize()] = ds.field[i].toByte();
      offset = offset + ds.field[i].getSize();
  }
  ```
- Send/receive the buffer to/from the RPC server
  ```java
  s = CreateSocket(IP, port);
  SendPacket(s, buf, buf.length());
  RecvPacket(s, buf, buf.length());
  ```
- Set parameters according to the buffer
  ```java
  offset = 100;
  for (int i=0; i<ds.getFieldNumber(); i++)
  {
      ds.field[i].setValue(buf, offset);
      offset = offset + ds.field[i].getSize();
  }
  ```

**Equivalent RPC Class**

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

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

```java
public class DS2 {
    DS1 x1[] = new DS1[100];
    DS2 *x2;
    ) DS2;
```
Testing J2C

Java Application

Call an RPC

Send/Recv()

C RPC Implementation

Call the RPC Implementation

Send/Recv()

TCP/UDP Socket

GetLocalTime(time, valid)

GetLocalOS(valid)