Remote Procedure Calls

RPC

- Introduced by Birrell & Nelson (1984)
- Remote Procedure Calls allow a program to make use of procedures executing on a remote machine
  - If it doesn't sound OO, it's because it isn't OO!
- RPCs are based on sockets, and therefore dispense us from using them directly
- Remote Procedures could in principle be written in a different language than the clients.

RPC Concepts

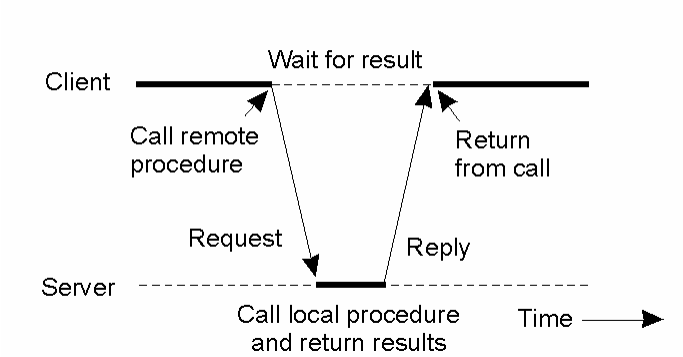
- Registration and identification of a procedure
  - The procedure registers its name to the RPC server
  - The client invokes the procedure by specifying the machine name, the procedure name, and by passing the parameters
- Encoding and decoding of procedure parameters
  - Given the difference in address space, and possibly in internal type encoding schemes and programming language, there is a need to use an external data representation
  - The client (and the server) will have to translate procedure parameters (encoding) and the return value (decoding) to and from that representation
  - Alternatively: have one side translate into the other side's representation, no common external representation (bad idea, why?)

RPC Concepts (cont.)

- Dealing with errors
  - Client and server can fail independently (not the case with "regular" procedure calls). How often should remote procedure be invoked?
    - Ideally: exactly once (hard to guarantee)
    - At-most once
    - At-least once (idempotent operations)
- Ordering relationships
  - Does order of two RPCs to same server matter?
  - If so, what should be that order (partial order semantics, total order semantics): Isis toolkit

Basic Communication Pattern

- Principle of RPC between a client and server program.

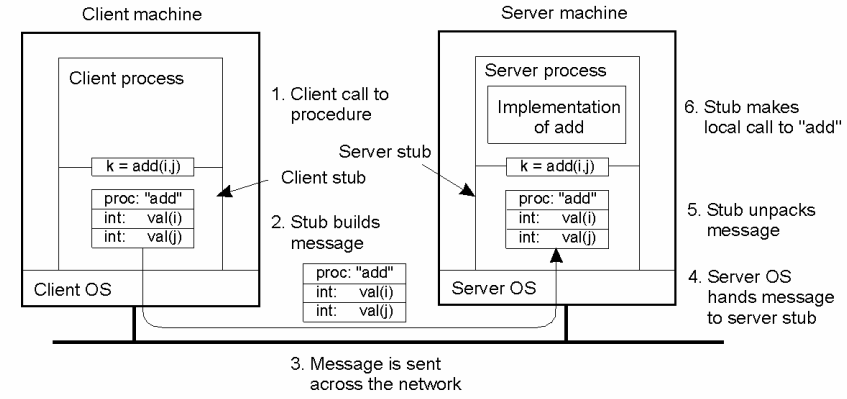


Steps of a Remote Procedure Call

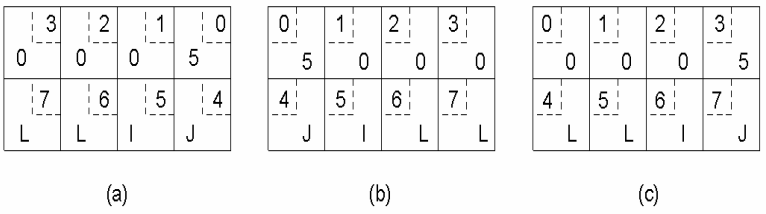
- Client procedure calls client stub in normal way
- Client stub builds message, calls local OS
- Client's OS sends message to remote OS
- Remote OS gives message to server stub
- Server stub unpacks parameters, calls server
- Server does work, returns result to the stub
- Server stub packs it in message, calls local OS
- Server's OS sends message to client's OS
- Client's OS gives message to client stub
- Stub unpacks result, returns to client

Passing Value Parameters (1)

- Steps involved in doing remote computation



Passing Value Parameters (2)



a) Original message on the Pentium
b) The message after receipt on the SPARC
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

RPC “old style”

- To use RPC in C under Unix…
- Each procedure belongs to a *program*
  - The program is designated by a number and version
  - The procedure itself has a number as well
- The `rpcinfo` command returns the list of procedures that are registered to the RPC server

RPC “old style”

- Example: (for the “portmapper” program)

```
Program  ver  proto    port
100000   2    tcp      111  portmapper
100000   2    udp      111  portmapper
```

- Obviously, each portmapper procedure has its own procedure number in addition.

RPC “old style”

- The intermediary representation “was” called eXternal Data Representation (XDR)
- XDR supports primitive data types and means to define more complex structures…
- Advantage: new computers (with new internal representations) need to convert to/from XDR only
- Disadvantage: requires typically two translations: from sender’s representation to XDR and then to receiver’s representation
- Alternative: Sender directly converts into receiver’s format (or vice versa). Disadvantages?

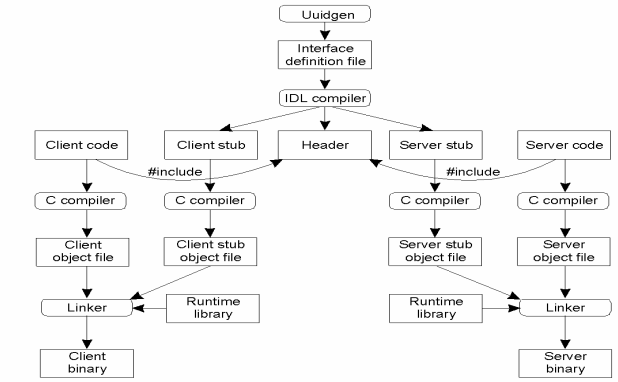
RPC “old style”

- Server-side:
  - Register the procedure:
    ```
    registerrpc(858993459, 1, 1, my_proc, xdr_pair, xdr_int)
    ```
  - Start listening to remote calls:
    ```
    svc_run()
    ```
- Client-side:
  - Call the procedure:
    ```
    test = callrpc(“Eureka”, 858993459, 1, 1, xdr_pair, &mypair,
    xdr_int, &myreturn);
    ```
  - Use the result!
    ```
    if (test == 0) printf(“return value: %d\n“, myreturn);
    ```

RPC “less old style”

- there has been since a way to hide XDR from the developer to some extent
- use of “stubs” and “skeletons” to do the encoding and decoding
  - we’ll see more of what those are when we cover RMI…
  - for the time being, check simple example on course website

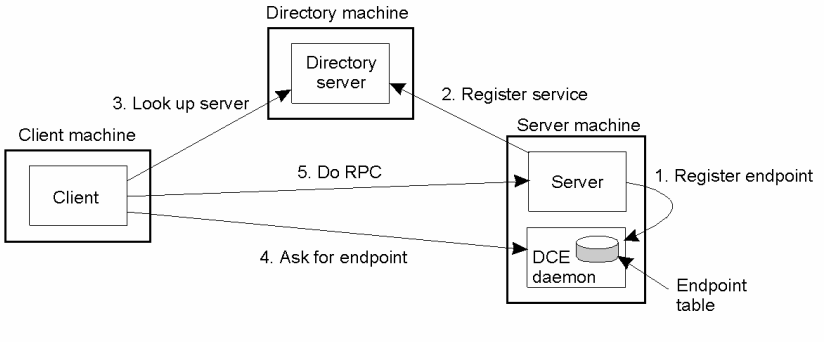
Writing a Client and a Server



- The steps in writing a client and a server in DCE RPC.

Binding a Client to a Server

- Client-to-server binding in DCE.



RPC "XML style": XML-RPC

- first released in 1998
- uses HTTP for transport (or SOAP, CORBA, …)
- XML replaces XDR as intermediate data representation
- Resources:
  - http://www.xmlrpc.com
  - http://xmlrpc-c.sourceforge.net/

XML-RPC data types

- XML-RPC supports these data types:
  - `base64 (for binary data)`
  - `boolean`
  - `date.Time.iso8601`
  - `double`
  - `int`
  - `string`
  - `struct (comparable to Hashtable)`
  - `array`

XML-RPC under the hood

- an XML-RPC client makes a remote procedure call using an HTTP POST request
- the following is a POST header example:

```
POST /XMLRPC HTTP/1.0
Host: www.advogato.org
Content-Type: text/xml
Content-Length: 151
```

XML-RPC under the hood

- after the header, the actual RPC call is encoded using XML
- the method is composed of the program name followed by the method (note: no brackets for methods)

```
<?xml version="1.0"?>
<methodCall>
   <methodName>test.square</methodName>
   <params>
      <param>
         <value><int>14</int></value>
      </param>
   </params>
</methodCall>
```

XML-RPC under the hood

- the XML-RPC response from the server is a standard HTTP response, with :

```
HTTP/1.0 200 OK
… (some header info omitted)
Content-Length: 157
Content-Type: text/xml
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><int>196</int></value>
      </param>
   </params>
</methodResponse>
```

Java support for XML-RPC

- without using any extra library:
  - you can always use the java.net.* facilities such as the URL and HttpUrlConnection classes to create XML-RPC requests and retrieve responses
  - you need to translate parameters into XML-RPC and parse the resulting XML
- or you can use for example the Apache XML-RPC class libraries:
  - java package org.apache.xmlrpc
  - http://xml.apache.org/xmlrpc

Apache XML-RPC Library

- client-side:
  - instantiate a client and specify the server address:

```
XmlRpcClient client = new
  XmlRpcClient("http://www.advogato.org");
```

  - store the remote procedure parameters in a `Vector`

```
Vector params = new Vector();
params.addElement(new Integer(14));
```

  - `execute` the call and get the return value in an `Object`

```
Integer result = (Integer) client.execute("test.square",
  params);
```

Apache XML-RPC Library

- client-side:
  - parameters must be of a type compatible to XML-RPC: `Boolean, byte[], Date, Double, Integer, String, Hashtable, Vector`
  - if there are no parameters, must still create an empty vector
  - the networking and all the XML encoding and decoding is handled by the XML-RPC library

Apache XML-RPC Library

- server-side:
  - use the simple built-in Web Server that only responds to XML-RPC requests:

```
WebServer server = new WebServer(4444);
```

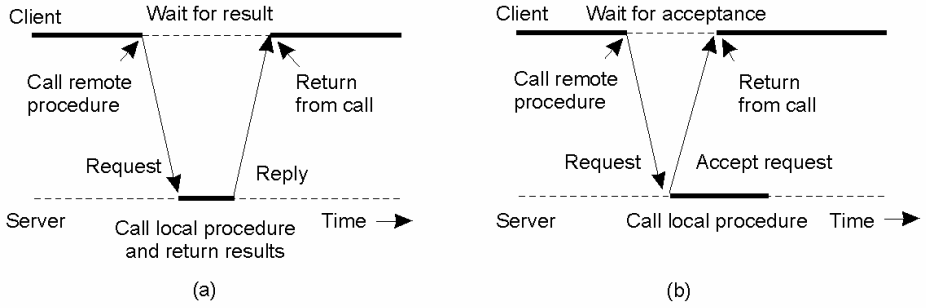
  - create the remote object, which methods will be called remotely

```
MyMathClass myMath = new MyMathClass();
```

- register the object with the server and give it a handler name:

```
server.addHandler("test", myMath);
```

Asynchronous RPC (1)

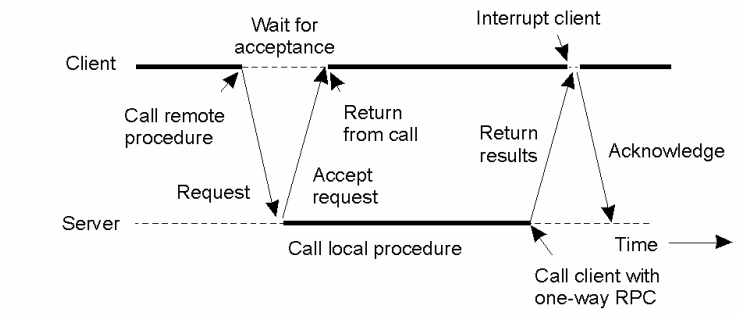


a) The interconnection between client and server in a traditional RPC

b) The interaction using asynchronous RPC

Asynchronous RPC (2)

- A client and server interacting through two asynchronous RPCs



Sun-RPC vs. XML-RPC

- Sun-RPC
  - Additional code gets generated (rpcgen)
  - More efficient
    - Binary messages: more compact, easier to generate
    - Uses sockets directly
  - Invocation of remote procedures using their names and parameters
- XML-RPC
  - No code generators
  - Message in human-readable form
  - More complex protocol stack: XML over HTTP/SOAP over TCP
  - Remote invocation somewhat generic

Final Thoughts on RPC

- makes networking easier:
  - hides the transport layer
  - no application-specific protocol to write
- makes networking slower:
  - because of all-purpose encoding/decoding
- must either conform to very simple parameter types, or must do some work
- enforces/limits application programmer to certain communication patterns