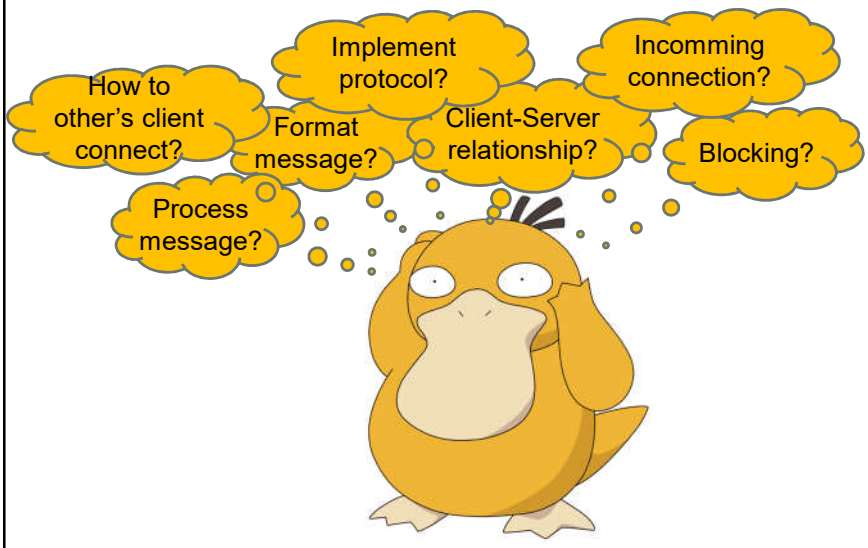
LEC 09.
RPC

Bui Trong Tung, SoICT, HUST

Content

- Introduction to RPC
- RPC Implemetation
- SUN ONC Framework
- Other RPC frameworks

How to resolve IT4062's HWs?



3

Remote Procedure Call (RPC)

- *The* most common framework for newer protocols and for middleware
- Used both by operating systems and by applications
  - NFS is implemented as a set of RPCs
  - DCOM, CORBA, Java RMI, etc., are just RPC systems
- Reference
  - Birrell, Andrew D., and Nelson, Bruce, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems,* vol. 2, #1, February 1984, pp 39-59.

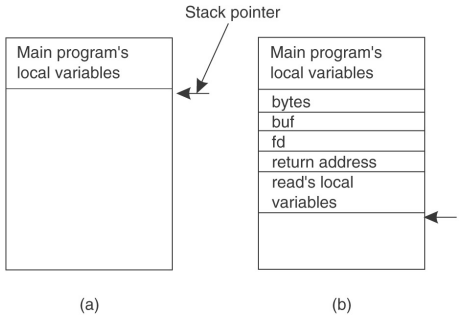
4

Fundamental idea

- RPC is modelled on the local procedure call, but the called procedure is executed in a different process and usually a different computer.
- Server process exports an *interface* of procedures or functions that can be called by client programs
  - similar to library API, class definitions, etc.
- Clients make local procedure/function calls
  - *As if* directly linked with the server process
  - Under the covers, procedure/function call is converted into a message exchange with remote server process

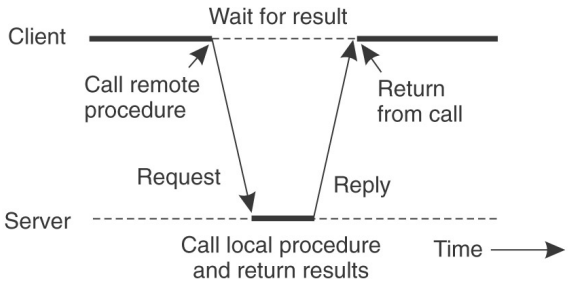
Ordinary procedure/function call

```
count = read(fd, buf, bytes)
```



Remote Procedure Call

- Would like to do the same if called procedure or function is on a remote server



7

Solution — a pair of *Stubs*

- A *client-side stub* is a function that looks to the client as if it were a callable service function
  - I.e., same API as the service's implementation of the function
- A *service-side stub* looks like a caller to the service
  - I.e., like a hunk of code invoking the service function
- The client program thinks it's invoking the service
  - but it's calling into the client-side stub
- The service program thinks it's called by the client
  - but it's really called by the service-side stub
- The stubs send messages to each other to make the RPC happen transparently (almost!)

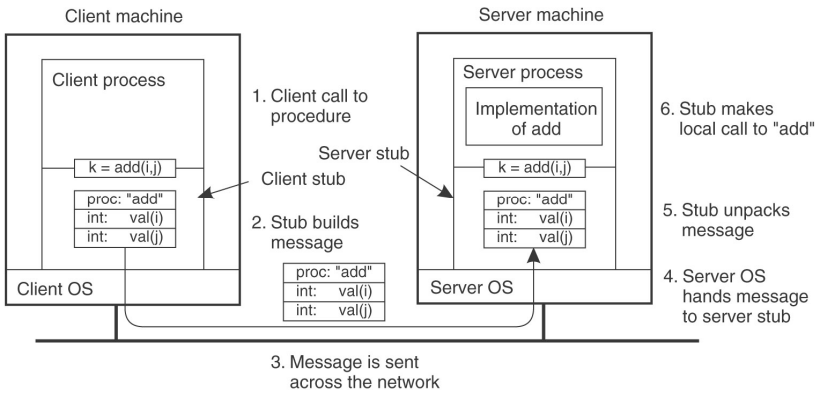
8

Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

9

RPC Stubs



10

RPC *Stubs* – Summary

- *Client-side stub*
  - Looks like local server function
  - Same interface as local function
  - Bundles arguments into a message, sends to server-side stub
  - Waits for reply, un-bundles results
  - returns
- *Server-side stub*
  - Looks like local client function to server
  - Listens on a socket for message from client stub
  - Un-bundles arguments to local variables
  - Makes a local function call to server
  - Bundles result into reply message to client stub

11

Result – a very useful Abstraction

- The hard work of building messages, formatting, uniform representation, etc., is buried in the stubs
  - Where it can be automated!
- Designers of client and server can concentrate on *semantics* of application
- Programs behave in familiar way

12

RPC vs LPC

- The called procedure is in another process which may reside in another machine.
- The processes do not share address space.
  - Passing of parameters by reference and passing pointer values are not allowed.
  - Parameters are passed by values.
- The called remote procedure executes within the environment of the server process.
  - The called procedure does not have access to the calling procedure's environment.

13

RPC – Issues

- How to handle failures?
- What are semantics of parameter passing?
  - E.g., pass by reference?
- How to bind (locate & connect) to servers?
- How to handle heterogeneity?
  - OS, language, architecture, …
- How to make it go fast?

14

Partial failures

- In local computing: if machine fails, application fails
- RPC failure:
  - Request from cli → srv lost
  - Reply from srv → cli lost
  - Server crashes after receiving request
  - Client crashes after sending request

→Partial failures

  - if a machine fails, part of application fails
  - one cannot tell the difference between a machine failure and network failure

15

Partial failures: Solution #1

- At-least-once call
  - With this call semantics, the client can assume that the remote procedure (RP) is executed at least once (on return from the RP).
  - Can be implemented by keep retrying on client side until you get a response.
  - Server just processes requests as normal, doesn't remember anything. Simple!
  - Acceptable only if the server's operations are idempotent. That is f(x) = f(f(x)).

16

Partial failures: Solution #2

- At-most-once call
  - When a RPC returns, it can assumed that the remote procedure has been called exactly once or not at all.
  - Implemented by the server's filtering of duplicate requests (which are caused by retransmissions due to IPC failure, slow or crashed server) and caching of replies
  - When the server crashes during the RP's execution, the partial execution may lead to erroneous results → the RP has not been executed at all

17

Partial failures: Solution #2

- At-most-once implementation:
  - Lost request message
  - Lost reply message
  - High latency
  - Server crashes after processing and lost reply message
  - Server crashes while processing

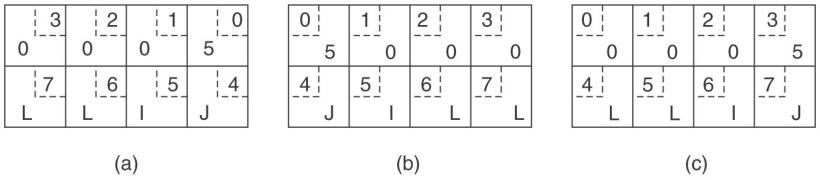
18

Marshalling Arguments

- *Marshalling* is the packing of function parameters into a message packet
  - the RPC stubs call type-specific functions to marshal or unmarshal the parameters of an RPC
    - Client stub marshals the arguments into a message
    - Server stub unmarshals the arguments and uses them to invoke the service function
  - on return:
    - the server stub marshals return values
    - the client stub unmarshals return values, and returns to the client program

Issue #1 — representation of data

- Big endian *vs.* little endian



(a) (b) (c)

Sent by Pentium Rec'd by SPARC After inversion

Issue #2 — Pointers and References

```
count = read(int fd, char* buf, int nbytes)
```

- Pointers are only valid within one address space
- Cannot be interpreted by another process
  - Even on same machine!
- Pointers and references are ubiquitous in C, C++
  - Even in Java implementations!

21

Pointers and References- Restricted Semantics

- Option: *call by value*
  - Sending stub dereferences pointer, copies result to message
  - Receiving stub conjures up a new pointer
- Option: *call by result*
  - Sending stub provides buffer, called function puts data into it
  - Receiving stub copies data to caller's buffer as specified by pointer
- Option: *call by value-result*
  - Caller's stub copies data to message, then copies result back to client buffer
  - Server stub keeps data in own buffer, server updates it; server sends data back in reply
- Not allowed:
  - *Call by reference*
  - *Aliased arguments*

22

RPC Implementation: IDL

- *Interface Definition Language*
- The IDL specifies the names, parameters, and types for all client-callable server procedures
- A *stub compiler* reads the IDL declarations and produces two *stub functions* for each server function
- IDL must also define representation of data on network
  - Multi-byte integers
  - Strings, character codes
  - Floating point, complex, …
  - …
- Clients and servers must *not* try to cast data

23

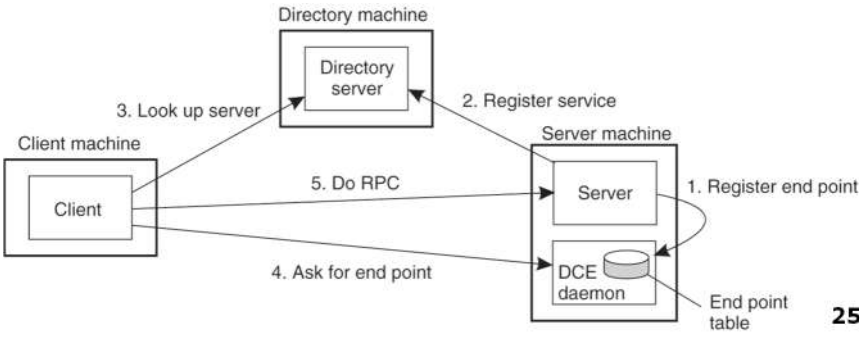
RPC Binding

- Binding is the process of connecting the client to the server
  - the server, when it starts up, exports its interface
    - identifies itself to a *network name server*
    - tells *RPC runtime* that it is alive and ready to accept calls
  - the client, before issuing any calls, imports the server
    - RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs

24

RPC Binding

- Registration of a server makes it possible for a client to locate the server and bind to it
- Server location is done in two steps:
  - Locate the server's machine.
  - Locate the server on that machine.



25

RPC framework

- DCE (Distributed Computing Environment)
  - Open Software Foundation
  - Basis for Microsoft DCOM
  - Tanenbaum & Van Steen, §4.2.4
- Sun's ONC (Open Network Computing)
  - Very similar to DCE
  - Widely used
  - **`rpcgen`**
  - http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/HTML/AA-Q0R5B-TET1_html/TITLE.html

26

RPC framework(cont.)

- Java RMI (Remote Method Invocation)
  - **`java.rmi`** standard package
  - Java-oriented approach — objects and methods
- CORBA (Common Object Request Broker Architecture)
  - Standard, multi-language, multi-platform middleware
  - Object-oriented
  - Heavyweight

27

Web RPC framework

- XML RPC
- SOAP
- Web Services và WSDL(VD: Apache CFX)
- .NET Web Services
- Web Service
- AJAX
- REST

28

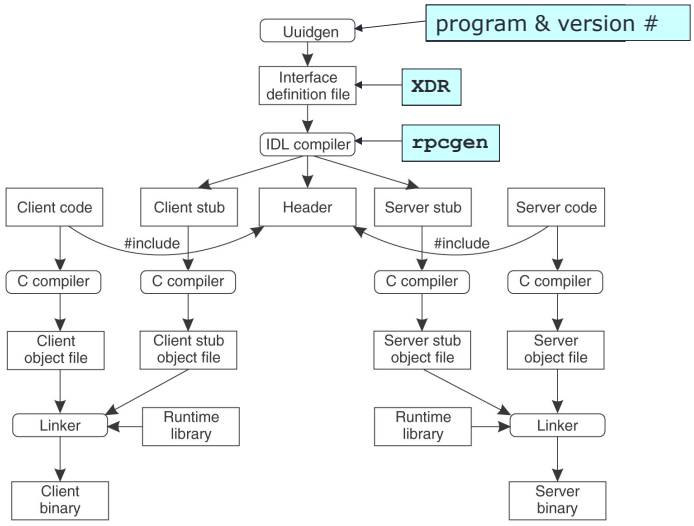
SUN ONC FRAMEWORK

Bui Trong Tung, SoICT, HUST

Sun ONC

- RPC for Unix System V, Linux, BSD, OS X
  - Also known as ONC RPC (Open Network Computing)
- Interfaces defined in an Interface Definition Language (IDL)
  - XDR: External Data Representation
  - IDL compiler is **rpcgen**

Implementation Model for ONC



31

Sun ONC: Steps

- Step 1: Make XDR file (.x)
- Step 2: Compiler .x file

```
$rpcgen -a -C .x
```

The results of compiling:

  - _clnt.c: client stub
  - _svc.c: server stub
  - _xdr: XDR file to marshalling
  - .h: define data type
  - Makefile.:
  - _client.c: client skeleton
  - _server.c: server skeleton
- Step 3: Compile to execuable files

32

.x file

- UUID: procedure ID
  - 0 - 1fffffff: used bySun
  - 20000000 - 3fffffff: can use
  - 40000000 - 5fffffff: temporary
  - 60000000 – ffffffff: reserved

```
/* Define data type of the arguments */
struct parameters {
    //Argument list
};
program APP_NAME {
   version APP_VERS {
       long PROCEDURE_1(parameters) = 1;
       string PROCEDURE_2(parameters) = 2;
   } = 1;   /* version */
} = UUID;
```

33

Sun ONC RPC – Binding

- Server: register **`portmapper`**
  - Program name
  - Version
  - Service port
- Client: call **`clnt_create()`**
  - Return: client handle to call server's procedure

```
CLIENT *clnt_create(
       char *host,          //Server address
       unsigned long prog, //UUID
       unsigned long vers, //Version
       char *proto          //Transport protocol
);
```

34

Sun ONC – Example

- Online tutorial
  - http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/HTML/AA-Q0R5B-TET1_html/TITLE.html
- Code samples
  - http://web.cs.wpi.edu/~rek/DCS/D04/SunRPC.html
  - http://web.cs.wpi.edu/~goos/Teach/cs4513-d05/
  - http://web.cs.wpi.edu/~cs4513/b05/week4-sunrpc.pdf
- Any other resources you can find

35

Sun ONC – Example

- Use Sun ONC to write the application that adds two number
- XDR file: add.x

```
struct intpair {
   int a;
   int b;
};
program ADD_PROG {
   version ADD_VERS {
       int ADD(intpair) = 1;
   } = 1;
} = 0x23451111;
```

- Compiler: `$rpcgen -a -C add.x`

36

Sun ONC – Example(cont)

- Edit add_server.c

```
/*
* insert server code here
*/
```

```
result = argp->a + argp->b;
printf("add(%d, %d) = %d\n", argp->a, argp->b, result);
```

37

Sun ONC – Example(cont)

- …edit add_client.c, then compiling

```
clnt = clnt_create (host, ADD_PROG, ADD_VERS, "udp");
```

```
clnt = clnt_create (host, ADD_PROG, ADD_VERS, "tcp");
```

```
result_1 = add_1(&add_1_arg, clnt);
if (result_1 == (int *) NULL) {
              clnt_perror (clnt, "call failed");
}
```

```
add_1_arg.a = 123;
add_1_arg.b = 22;
result_1 = add_1(&add_1_arg, clnt);
if (result_1 == (int *) NULL)
             clnt_perror (clnt, "call failed");
else printf("result = %d\n", *result_1);
```

38