Remote Procedure Calls & Remote Objects

527950-1
Fall 2019
10/17/2019
Kyoung Shin Park
Applied Computer Engineering
Dankook University

Overview

- Remote Procedure Call
- Remote Objects
- Web Services
- Marshaling

RPC

- Remote Procedure Call (RPC) for Unix system V, Linux, BSD, OS X
  - ONC (Open Network Computing)
  - Created by Sun
  - RFC 1831 (1995), RFC 5531 (2009)
  - Remains in use mostly because of **NFS (Network File System)**
    - **NFS is implemented as a set of RPCs**
- Interfaces defined in an **Interface Definition Language (IDL)**
- IDL compiler is ***rpcgen***

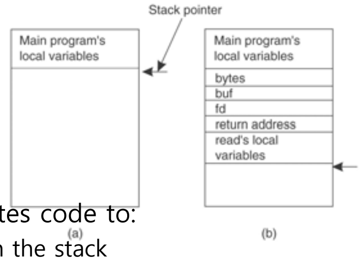
RPC

- Problems with sockets API
  - The **sockets** interface forces a **read/write** mechanism
  - Programming is often easier with a **functional interface**
- **RPC (Remote Procedure Call)**
  - Mechanism to **call procedures on a remote machine**

RPC

- Fundamental idea
  - 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/functional call is converted into a message exchange with remote server process

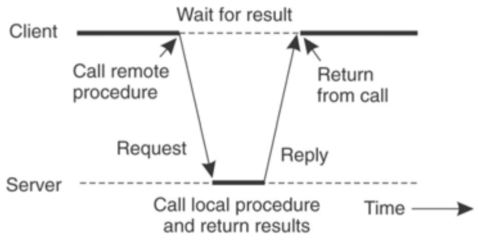
RPC



- In regular procedure calls,
  - **count = read(fd, buf, nbytes);**
  - The compiler parse this and generates code to:
    - Push the current value of **nbytes** on the stack
    - Push the address of **buf** on the stack
    - Push the value of **fd** on the stack
    - Generate a call to the function **read**
  - In compiling f, the compiler generates code to:
    - Push registers that will be clobbered on the stack to **save the values**
    - **Adjust the stack** to make room for local and temporary variables
    - Before a return, un-adjust the stack, put the **return data** in a register, and **issue a return instruction**

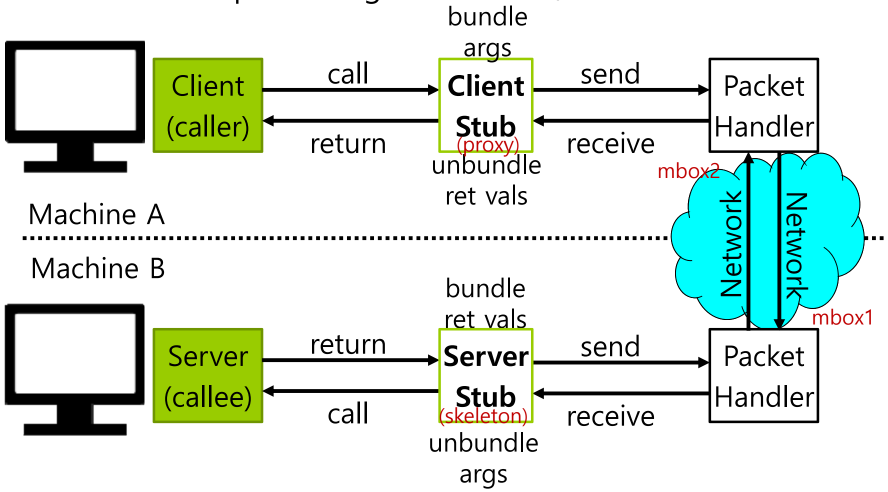
RPC

- Do the same if called procedure is on a remote server
- Equivalence with regular procedure call
  - Parameters <-> Request message
  - Result <-> Reply message
  - Name of procedure: Passed in request message
  - Return address: mbox2 (client return mail box)



RPC

- RPC "Stub" provides glue on client/server



RPC

- **Client-side stub**
  - Looks like a local server function
  - Same interface as local function
  - **Bundles arguments into message**, sends to server-side stub (marshaling)
  - Waits for **reply**, **unbundles results** (unmarshaling)
  - **Returns** to client code
- **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** (unmarshaling)
  - Makes **a local function call** to server
  - **Bundles result into reply message** to client stub (marshaling)

RPC Proxy

- A client-side stub (proxy) looks like the remote function
  - Client stub has the same interface as the remote function
  - Looks & feels like the remote function to the programmer, but its function is to:
    - Marshal parameters
    - Send the message
    - Wait for a response from the server
    - Unmarshal the response & return the appropriate data
    - Generate exceptions if problems arise

RPC Skeleton

- A server-side stub (skeleton) is really two parts:
  - **Dispatcher**
    - Receives client requests
    - Identifies appropriate functions
  - **Skeleton**
    - Unmarshals parameters
    - Calls the local server procedure
    - Marshals the responses & sends it back to the dispatcher

RPC

- Implementation issues
  - **The hard work of building messages, formatting, uniform representation, etc.,** is buried in the stubs
  - Client and server designers can concentrate on the **semantics** of application
  - How to make the "remote" part of RPC invisible to the programmer?
  - 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?

RPC Models

- A server defines the **service interface** using an **interface definition language (IDL)**
  - **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
  - **Server-side and client-side**
- Linking
  - Server programmer implements the service's functions and links with the server-side stubs
  - Client programmer implements the client program and links it with client-side stubs
- Operation
  - Stubs manage all of the details of remote communication between client and server

RPC Stubs

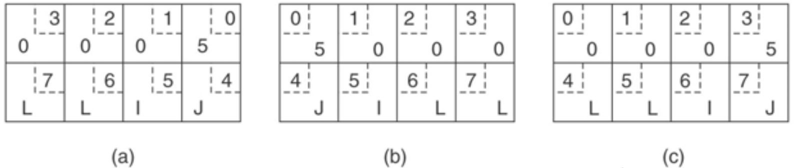
- A client-side stub is a function that looks to the client as if it were a callable server function
  - I.e., same API as the server's implementation of the function
- A server-side stub looks like a caller to the server
  - I.e., like a hunk of code invoking the server function
- The client program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

RPC Marshalling

- **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-side stub marshals the arguments into a message
  - Server-side stub unmarshals the arguments and uses them to invoke the service function
- On return:
  - The server-side stub marshals return values
  - The client-side stub unmarshals return values, and returns to the client program

RPC Issue – Representing Data

- **Big endian vs Little endian**
  - Big endian – Most significant byte in low memory
  - Little endian - Most significant byte in high memory



(a) Sent by Pentium

(b) Rec'd by SPARC

(c) After inversion

- IDL must also define representation of data on network
  - **Multi-byte integers, strings, character codes, floating point,..**
- Each stub converts machine representation to/from network representation
- Clients and servers must not try to cast data!

RPC Issue – Representing Data

- Serialization
  - Need **standard** encoding to enable communication between **heterogeneous** systems
  - Serialization convert data into a pointerless format: **an array of bytes**
  - Examples:
    - XDR (eXternal Data Representation), used by ONC RPC
    - JSON (JavaScript Object Notation)
    - W3C XML Schema Language
    - ASN.1 (ISO Abstract Syntax Notation)
    - Google Protocol Buffers
  - Implicit vs Explicit typing
    - Implicit typing - **only values are transmitted**, not data types or parameter information, e.g. ONC **XDR** (RFC 4506)
    - Explicit typing – **type is transmitted with each value**, e.g. ISO's **ASN.1, JSON, XML, protocol buffers**

RPC Issue – Pointers and References

- **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!

RPC Issue – Pointers and References

- Option: call by value (copy data to network message)
  - 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
  - Client 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 or aliased arguments

RPC Binding

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other information)
- **Binding** is the process of connecting the client to the server
  - The **server**, when it starts up, **exports its interface**
    - The server identifies itself to a network **name server**
    - The 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

Remote Objects

- Microsoft COM+ (DCOM)
  - Unified COM and DCOM plus support for transactions, resource pooling, publish-subscribe communication
  - Extends Component Object Model (COM) to allow objects to communicate between machines

Java RMI

- Java language had no mechanism for invoking remote methods
- 1995 Sun added extension
  - Remote Method Invocation (RMI)
  - Allow programmer to create distributed applications where methods of remote objects can be invoked from other JVMs

Java RMI

- Client
  - Invokes method on remote object
- Server
  - Process that owns the remote object
- Object registry
  - Name server that relates objects with names

Marshalling

- Standard formats for data
  - Network Data Representation (NDR)
- Goal
  - Multi-canonical approach to data conversion
  - Fixed set of alternate representations
  - Byte order, character sets, and floating-point representation can assume one of several forms
  - Sender can (hopefully) use native format
  - Receiver may have to convert

Reference

- http://www.cs.colostate.edu/~cs551/CourseNotes/Processes/ProcessTOC.html
- http://web.cs.wpi.edu/~cs4513/d07/LectureNotes/Week%201%20--%20Remote%20Procedure%20Call.ppt