Distributed Systems
8. Remote Procedure Calls

Paul Krzyzanowski
pxk@cs.rutgers.edu
Problems with the sockets API

The **sockets** interface forces a read/write mechanism

Programming is often easier with a functional interface

To make distributed computing look more like centralized computing, I/O (read/write) is not the way to go
Remote Procedure Call

Goal: it should appear to the programmer that a normal call is taking place
Regular procedure calls

You write:

\[ x = f(a, \text{"test"}, 5); \]

The compiler parses this and generates code to:

a. Push the value 5 on the stack
b. Push the address of the string “test” on the stack
c. Push the current value of a on the stack
d. Generate a call to the function f

In compiling \( f \), the compiler generates code to:

a. Push registers that will be clobbered on the stack to save the values
b. Adjust the stack to make room for local and temporary variables
c. Before a return, unadjust the stack, put the return data in a register, and issue a return instruction
Implementing RPC

No architectural support for remote procedure calls

*Simulate it* with tools we have
(local procedure calls)

Simulation makes RPC a
language-level construct

instead of an
operating system construct

The compiler creates code to
send messages to
invoke remote functions

The OS gives us
sockets
Implementing RPC

The trick:

Create **stub functions** to make it appear to the user that the call is local

The stub function contains the function’s interface
Stub functions

1. Client calls stub (params on stack)

Client functions

client stub

network routines

Client

Server functions

server stub (skeleton)

network routines

Server
2. Stub marshals params to net message

Marshalling = put data in a form suitable for transmission over a network (serialized)
3. Network message sent to server

Client functions

Client stub

Network routines

Server stub (skeleton)

Network routines

Server functions

© 2012 Paul Krzyzanowski
4. Receive message: send it to server stub
5. Unmarshal parameters, call server function

Client functions

client stub

network routines

Server functions

server stub (skeleton)

network routines
6. Return from server function
Stub functions

7. Marshal return value and send message
8. Transfer message over network

```
client functions
  ↓
client stub
  ↓
network routines
```

```
server functions
  ↑
server stub (skeleton)
  ↑
network routines
```

client

server
Stub functions

9. Receive message: client stub is receiver

client functions

client stub

network routines

server functions

server stub (skeleton)

network routines

client

server

© 2012 Paul Krzyzanowski
Stub functions

10. Unmarshal return value, return to client code
Benefits

• Procedure call interface

• Writing applications is simplified
  – RPC hides all network code into stub functions
  – Application programmers don’t have to worry about details
    • Sockets, port numbers, byte ordering

• Where is RPC in the OSI model?
  – Layer 5: Session layer: Connection management
  – Layer 6: Presentation: Marshaling/data representation
RPC has challenges
Parameter passing

Pass by value
  – Easy: just copy data to network message

Pass by reference
  – Makes no sense without shared memory
Pass by reference?

1. Copy items referenced to message buffer
2. Ship them over
3. Unmarshal data at server
4. Pass *local* pointer to server stub function
5. Send new values back

To support complex structures
– Copy structure into pointerless representation
– Transmit
– Reconstruct structure with local pointers on server
Representing data

No such thing as
**incompatibility problems** on local system

Remote machine may have:
- Different byte ordering
- Different sizes of integers and other types
- Different floating point representations
- Different character sets
- Alignment requirements
Representing data

IP (headers) forced all to use **big endian** byte ordering for 16- and 32-bit values

**Big endian**: Most significant byte in low memory
- SPARC ≤ V9, Motorola 680x0, older PowerPC

**Little endian**: Most significant byte in low memory
- Intel IA-32, x64

**Bi-endian**: Processor may operate in either mode
- ARM, PowerPC, MIPS, SPARC V9, IA-64 (Intel Itanium)

```c
main() {
    unsigned int n;
    char *a = (char *)&n;

    n = 0x11223344;
    printf("%02x, %02x, %02x, %02x
",
            a[0], a[1], a[2], a[3]);
}
```

Output on an Intel:
44, 33, 22, 11

Output on a PowerPC:
11, 22, 33, 44
Representing data

Need standard encoding to enable communication between heterogeneous systems

- e.g. Sun’s RPC uses XDR (eXternal Data Representation)
- ASN.1 (ISO Abstract Syntax Notation)
- JSON (JavaScript Object Notation)
- Google Protocol Buffers
- W3C XML Schema Language
Representing data

**Implicit typing**
- only values are transmitted, not data types or parameter info
- e.g., Sun XDR

**Explicit typing**
- Type is transmitted with each value
- e.g., ISO’s ASN.1, XML, protocol buffers, JSON
Where to bind?

Need to locate host and correct server process
Where to bind? – Solution 1

Maintain centralized DB that can locate a host that provides a particular service

(Birrell & Nelson’s 1984 proposal)
Where to bind? – Solution 2

A server on each host maintains a DB of *locally* provided services

Solution 1 is problematic for Sun NFS – identical file servers serve different file systems
Transport protocol

TCP or UDP? Which one should we use?

• Some implementations may offer only one (e.g. TCP)

• Most support several
  – Allow programmer (or end user) to choose at runtime
When things go wrong

• Local procedure calls do not fail
  – If they core dump, entire process dies

• More opportunities for error with RPC

• Transparency breaks here
  – Applications should be prepared to deal with RPC failure
When things go wrong

• **Semantics of remote procedure calls**
  – Local procedure call: *exactly once*

• A remote procedure call may be called:
  – 0 times:
    server crashed or server process died before executing server code
  – 1 time:
    everything worked well, as expected
  – 1 or more: excess latency or lost reply from server and client retransmission
RPC semantics

• Most RPC systems will offer either:
  – *at least once* semantics
  – or *at most once* semantics

• Understand application:
  – *idempotent* functions: may be run any number of times without harm
  – *non-idempotent* functions: those with side-effects
More issues

**Performance**

- RPC is slower … a lot slower

**Security**

- messages visible over network
- Authenticate client
- Authenticate server
Programming with RPC

Language support

– Most programming languages (C, C++, Java, …) have no concept of remote procedure calls
– Language compilers will not generate client and server stubs

Common solution:
– Use a separate compiler to generate stubs (pre-compiler)
Interface Definition Language

• Allow programmer to specify remote procedure interfaces (names, parameters, return values)

• Pre-compiler can use this to generate client and server stubs:
  – Marshaling code
  – Unmarshaling code
  – Network transport routines
  – Conform to defined interface

• Similar to function prototypes
RPC compiler

IDL

RPC compiler

client code (main)

client stub

data conv.

headers

data conv.

server skeleton

server functions

compiler

client

server

Code you write

Code RPC compiler generates
Writing the program

• Client code has to be modified
  – Initialize RPC-related options
    • Transport type
    • Locate server/service
  – Handle failure of remote procedure call

• Server functions
  – Generally need little or no modification
RPC API

What kind of services does an RPC system need?

• **Name service operations**
  – Export/lookup of binding information (ports, machines)
  – Support dynamic ports

• **Binding operations**
  – Establish client/server communications using appropriate protocol (establish endpoints)

• **Endpoint operations**
  – Listen for requests, export endpoint to name server (often the *main* program on the server)
What kind of services does an RPC system need?

• **Security operations**
  – Authenticate client/server

• **Internationalization operations** (possibly)

• **Marshaling/data conversion operations**

• **Stub memory management**
  – Dealing with “reference” data, temporary buffers

• **Program ID operations**
  – Allow applications to access IDs of RPC interfaces
  – Can you pass references to remote functions to other processes?
The End