

Distributed Systems

03. Remote Procedure Calls

Paul Krzyzanowski
Rutgers University
Fall 2018

September 17, 2018 © 2014-2018 Paul Krzyzanowski 1

Socket-based communication

- Socket API: all we get from the OS to access the network
- Socket = distinct end-to-end communication channels
 - Read/write model
- Line-oriented, text-based protocols common
 - Not efficient but easy to debug & use

September 17, 2018 © 2014-2018 Paul Krzyzanowski 2

Sample SMTP Interaction

SMTP = Simple Mail Transfer Protocol

```

$ telnet aramis.rutgers.edu 25
Trying 128.6.4.2...
Connected to aramis.rutgers.edu.
Escape character is '^]'.
220 aramis.rutgers.edu ESMTP Sendmail 8.11.7p3+Sun/8.8.8; Sun, 17 Sep 2017 23:25:41 -0400 (EDT)
HELO pk.org
250 aramis.rutgers.edu Hello aramis.rutgers.edu [128.6.4.2], pleased to meet you
MAIL FROM: <pxk@cs.rutgers.edu>
250 2.1.0 <pxk@cs.rutgers.edu>... Sender ok
RCPT TO: <p@pk.org>
250 2.1.5 <p@pk.org>... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
From: Paul Krzyzanowski <pxk@cs.rutgers.edu>
Subject: test message
Date: Mon, 17 Feb 2020 17:00:16 -0500
To: Whomever <testuser@pk.org>

Hi,
This is a test

250 2.0.0 v8l3QY124658 Message accepted for delivery
quit
221 2.0.0 aramis.rutgers.edu closing connection
    
```

This is the message body. Headers may define the structure of the message but are ignored for delivery.

September 17, 2018 © 2014-2018 Paul Krzyzanowski 3

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

September 17, 2018 © 2014-2018 Paul Krzyzanowski 4

RPC

1984: Birrell & Nelson

- Mechanism to call procedures on other machines

Remote Procedure Call

September 17, 2018 © 2014-2018 Paul Krzyzanowski 5

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

September 17, 2018 © 2014-2018 Paul Krzyzanowski 6

Implementing RPC

The trick:

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

On the client

- The stub function (**proxy**) has the function's interface
- Packages parameters and calls the server

On the server

- The stub function (**skeleton**) receives the request and calls the local function

September 17, 2018 © 2014-2018 Paul Krzyzanowski 7

Stub functions

1. Client calls stub (params on stack)

September 17, 2018 © 2014-2018 Paul Krzyzanowski 8

Stub functions

2. Stub **marshals** params to network message

Marshalling = put parameters in a form suitable for transmission over a network (serialized)

September 17, 2018 © 2014-2018 Paul Krzyzanowski 9

Stub functions

3. Network message sent to server

September 17, 2018 © 2014-2018 Paul Krzyzanowski 10

Stub functions

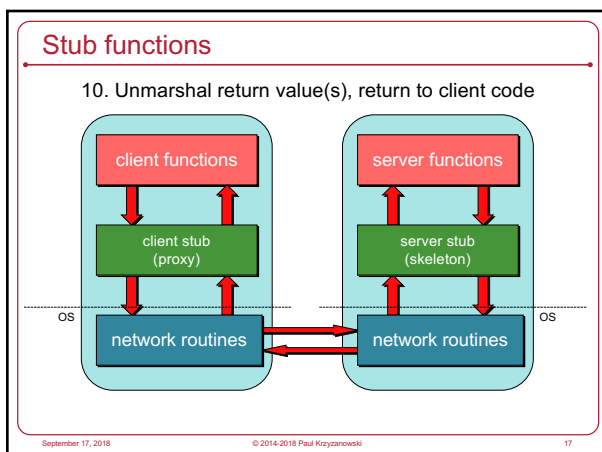
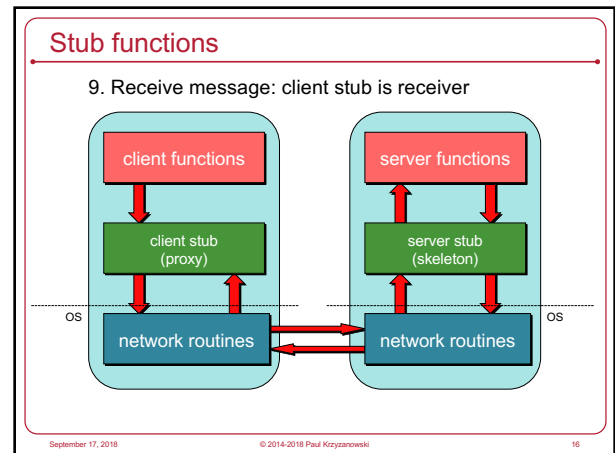
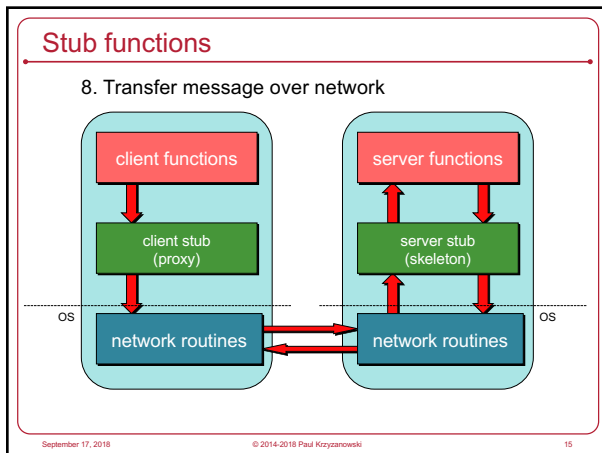
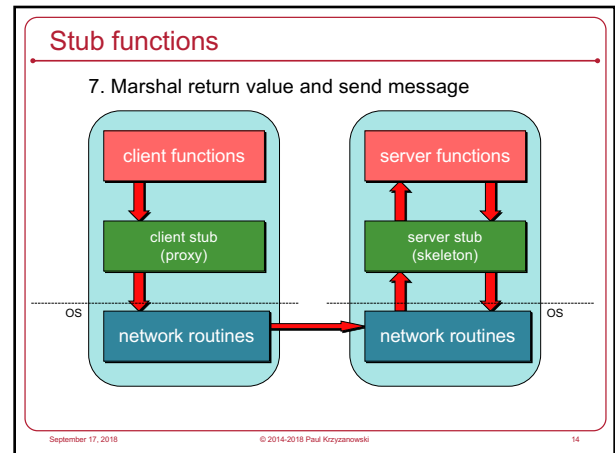
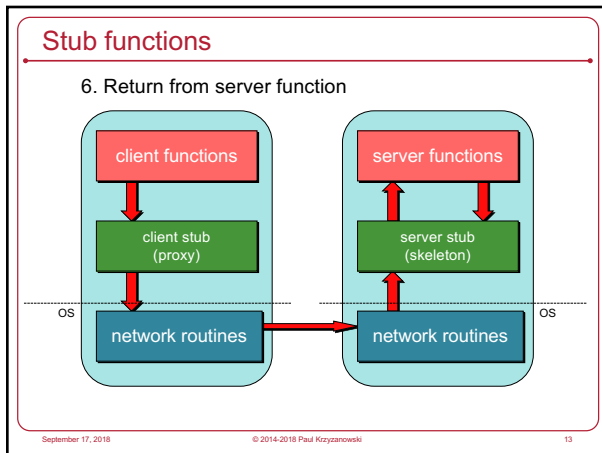
4. Receive message: send it to server stub

September 17, 2018 © 2014-2018 Paul Krzyzanowski 11

Stub functions

5. Unmarshal parameters, call server function

September 17, 2018 © 2014-2018 Paul Krzyzanowski 12



A client 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

September 17, 2018 © 2014-2018 Paul Krzyzanowski 18

A server stub contains two parts

- **Dispatcher – the listener**
 - Receives client requests
 - Identifies appropriate function (method)
- **Skeleton – the unmarshaller & caller**
 - Unmarshals parameters
 - Calls the local server procedure
 - Marshals the response & sends it back to the dispatcher
- All this is invisible to the programmer
 - The programmer doesn't deal with any of this
 - Dispatcher + Skeleton may be integrated
 - Depends on implementation

September 17, 2018

© 2014-2018 Paul Krzyzanowski

19

RPC Benefits

- RPC gives us a 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
 - Uses the transport layer (4) for communication (TCP/UDP)

September 17, 2018

© 2014-2018 Paul Krzyzanowski

20

RPC has challenges

September 17, 2018

© 2014-2018 Paul Krzyzanowski

21

RPC Issues

- **Parameter passing**
 - Pass by value or pass by reference?
 - **Pointerless** representation
- **Service binding**. How do we locate the server endpoint?
 - Central DB
 - DB of services per host
- **Transport protocol**
 - TCP? UDP? Both?
- **When things go wrong**
 - Opportunities for failure

September 17, 2018

© 2014-2018 Paul Krzyzanowski

22

When things go wrong

- **Semantics of remote procedure calls**
 - Local procedure call: *exactly once*
- Most RPC systems will offer either
 - **at least once** semantics
 - or **at most once** semantics
- Decide based on application
 - **idempotent** functions: may be run any number of times without harm
 - **non-idempotent** functions: those with side-effects
- Ideally – design your application to be idempotent
 - ... and stateless
 - Not always easy!
 - Store transaction IDs, previous return data, etc.

September 17, 2018

© 2014-2018 Paul Krzyzanowski

23

More issues

- **Performance**
 - RPC is slower ... a lot slower (why?)
- **Security**
 - messages may be visible over network – do we need to hide them?
 - Authenticate client?
 - Authenticate server?

September 17, 2018

© 2014-2018 Paul Krzyzanowski

24

Programming with RPC

Language support

- Many programming languages have no language-level concept of remote procedure calls (C, C++, Java <J2SE 5.0, ...)
- These compilers will not automatically generate client and server stubs
- Some languages have support that enables RPC (Java, Python, Haskell, Go, Erlang)
- But we may need to deal with heterogeneous environments (e.g., Java communicating via XML)

Common solution

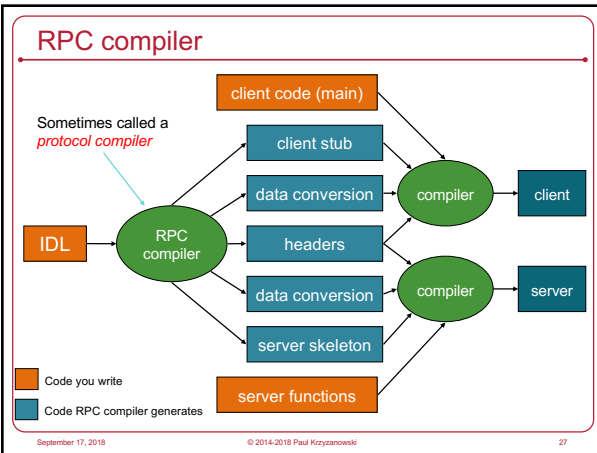
- Interface Definition Language (IDL): describes remote procedures
- Separate compiler that generate stubs (pre-compiler)

September 17, 2018 © 2014-2018 Paul Krzyzanowski 25

Interface Definition Language (IDL)

- 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
- An IDL looks similar to function prototypes

September 17, 2018 © 2014-2018 Paul Krzyzanowski 26



Writing the program

- Client code has to be modified
 - Initialize RPC-related options
 - Identify transport type
 - Locate server/service
 - Handle failure of remote procedure calls
- Server functions
 - Generally need little or no modification

September 17, 2018 © 2014-2018 Paul Krzyzanowski 28

Sending data over the network

September 17, 2018 © 2014-2018 Paul Krzyzanowski 29

Stream of bytes

```

struct item {
    char name[64];
    unsigned long id;
    int number_in_stock;
    float rating;
    double price;
} scratcher = {
    "Bear Claw Black Telescopic Back Scratcher",
    00120,
    332,
    4.6,
    5.99
}
42 65 61 72 20 43 6c 61 77 20 42 6c 61 63 6b 20 54 ...
    
```

September 17, 2018 © 2014-2018 Paul Krzyzanowski 30

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

September 17, 2018 © 2014-2018 Paul Krzyzanowski 31

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 ← IP headers use big endian

Little endian: Most significant byte in high memory
 - Intel/AMD IA-32, x64

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

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

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

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

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

September 17, 2018 © 2014-2018 Paul Krzyzanowski 32

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

September 17, 2018 © 2014-2018 Paul Krzyzanowski 33

Serializing data

Implicit typing

- only values are transmitted, not data types or parameter info
- e.g., ONC XDR (RFC 4506)

Explicit typing

- Type is transmitted with each value
- e.g., ISO's ASN.1, XML, protocol buffers, JSON

Marshaling vs. serialization – almost synonymous:

Serialization: converting an object into a sequence of bytes that can be sent over a network

Marshaling: bundling parameters into a form that can be reconstructed (unmarshaled) by another process. May include object ID or other state. Marshaling uses serialization.

September 17, 2018 © 2014-2018 Paul Krzyzanowski 34

XML: eXtensible Markup Language

```
<ShoppingCart>
  <Items>
    <Item>
      <ItemID> 00120 </ItemID>
      <Item> Bear Claw Black Telescopic Back Scratcher </Item>
      <Price> 5.99 </Price>
    </Item>
    <ItemID> 00121 </ItemID>
    <Item> Scalp Massager </Item>
    <Price> 5.95 </Price>
  </Items>
</ShoppingCart>
```

Benefits:

- Human-readable
- Human-editable
- Interleaves structure with text (data)

September 17, 2018 © 2014-2018 Paul Krzyzanowski 35

JSON: JavaScript Object Notation

- Lightweight (relatively efficient) data interchange format
 - Introduced as the *"fat-free alternative to XML"*
 - Based on JavaScript
- Human writeable and readable
- Self-describing (explicitly typed)
- Language independent
- Easy to parse
- Currently converters for 50+ languages
- Includes support for RPC invocation via JSON-RPC

September 17, 2018 © 2014-2018 Paul Krzyzanowski 36

JSON Example

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menutitem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" },
        { "value": "Close", "onclick": "CloseDoc()" }
      ]
    }
  }
}
```

from json.org/example.html

September 17, 2018

© 2014-2018 Paul Krzyzanowski

37

Google Protocol Buffers

- Efficient mechanism for serializing structured data
 - Much simpler, smaller, and faster than XML
- Language independent
- Define messages
 - Each message is a set of names and types
- Compile the messages to generate data access classes for your language
- Used extensively within Google. Currently over 48,000 different message types defined.
 - Used both for RPC and for persistent storage

September 17, 2018

© 2014-2018 Paul Krzyzanowski

38

Example (from the Developer Guide)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

September 17, 2018

© 2014-2018 Paul Krzyzanowski

39

Example (from the Developer Guide)

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

September 17, 2018

© 2014-2018 Paul Krzyzanowski

40

Efficiency example (from the Developer Guide)

```
<person>
  <name>John Doe</name>
  <email>jdoe@example.com</email>
</person>
```

XML version

```
person {
  name: "John Doe"
  email: "jdoe@example.com"
}
```

Text (uncompiled) protocol buffer

- Binary encoded message: ~28 bytes long, 100-200 ns to parse
- XML version: ≥69 bytes, 5,000-10,000 ns to parse

September 17, 2018

© 2014-2018 Paul Krzyzanowski

41

The End

September 17, 2018

© 2014-2018 Paul Krzyzanowski

42