Week 2: Part 2
Data Encoding in Remote Procedure Calls
Sending data over the network
struct item {
    char name[64];
    unsigned long id;
    int number_in_stock;
    float rating;
    double price;
}

scratch = {
    "Bear Claw Black Telescopic Back Scratcher",
    00120,
    332,
    4.6,
    5.99
}

gets stored in memory as:

```
42 65 61 72 20 43 6c 61 77 20 42 6c 61 63 6b 20 54 ...
```
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
IP (headers) forced all to use **big endian** byte ordering for 16- and 32-bit values

**Big endian**: Most significant byte in low memory
- Java Virtual Machine, OpenRISC, Atmel AVR32, IBM z-series, SPARC < V9, Motorola 680x0, older PowerPC

**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)

```c
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
Representing data: serialization

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

**Benefits:**
- Human-readable
- Human-editable
- Interleaves structure with text (data)

**Problems:**
-Verbose: transmit more data than needed
- Longer parsing time
- Data conversion always required for numbers
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
```json
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuitem": [
        {"value": "New", "onclick": "CreateNewDoc()"},
        {"value": "Open", "onclick": "OpenDoc()"},
        {"value": "Close", "onclick": "CloseDoc()"}
      ]
    }
  }
}
```
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
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;
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);
Efficiency example (from the Developer Guide)

- Binary encoded message: ~28 bytes long, 100-200 ns to parse
- XML version: ≥ 69 bytes, 5,000-10,000 ns to parse
- In general,
  - 3-10x smaller data
  - 20-100 times faster to marshal/unmarshal
  - Easier to use programmatically

XML version

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

Text (uncompiled) protocol buffer

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

http://code.google.com/apis/protocolbuffers/docs/overview.html
The End