# CSCI-1680
# RPC and Data Representation

Rodrigo Fonseca

# Today

- **Defining Protocols**
  - RPC
  - IDL

# Problem

- **Two programs want to communicate: must define the protocol**
  - We have seen many of these, across all layers
  - E.g., Snowcast packet formats, protocol headers
- **Key Problems**
  - Semantics of the communication
    - APIs, how to cope with failure
  - Data Representation
  - Scope: should the scheme work across
    - Architectures
    - Languages
    - Compilers…?

# RPC – Remote Procedure Call

- **Procedure calls are a well understood mechanism**
  - Transfer control and data on a single computer

- **Idea: make distributed programming look the same**
  - Have servers export interfaces that are accessible through local APIs
  - Perform the illusion behind the scenes

- **2 Major Components**
  - Protocol to manage messages sent between client and server
  - Language and compiler support
    - Packing, unpacking, calling function, returning value

# Stub Functions

- **Local stub functions at client and server give appearance of a local function call**
- **client stub**
  - marshalls parameters -> sends to server -> waits
  - unmarshalls results -> returns to client
- **server stub**
  - creates socket/ports and accepts connections
  - receives message from client stub -> unmarshalls parameters -> calls server function
  - marshalls results -> sends results to client stub
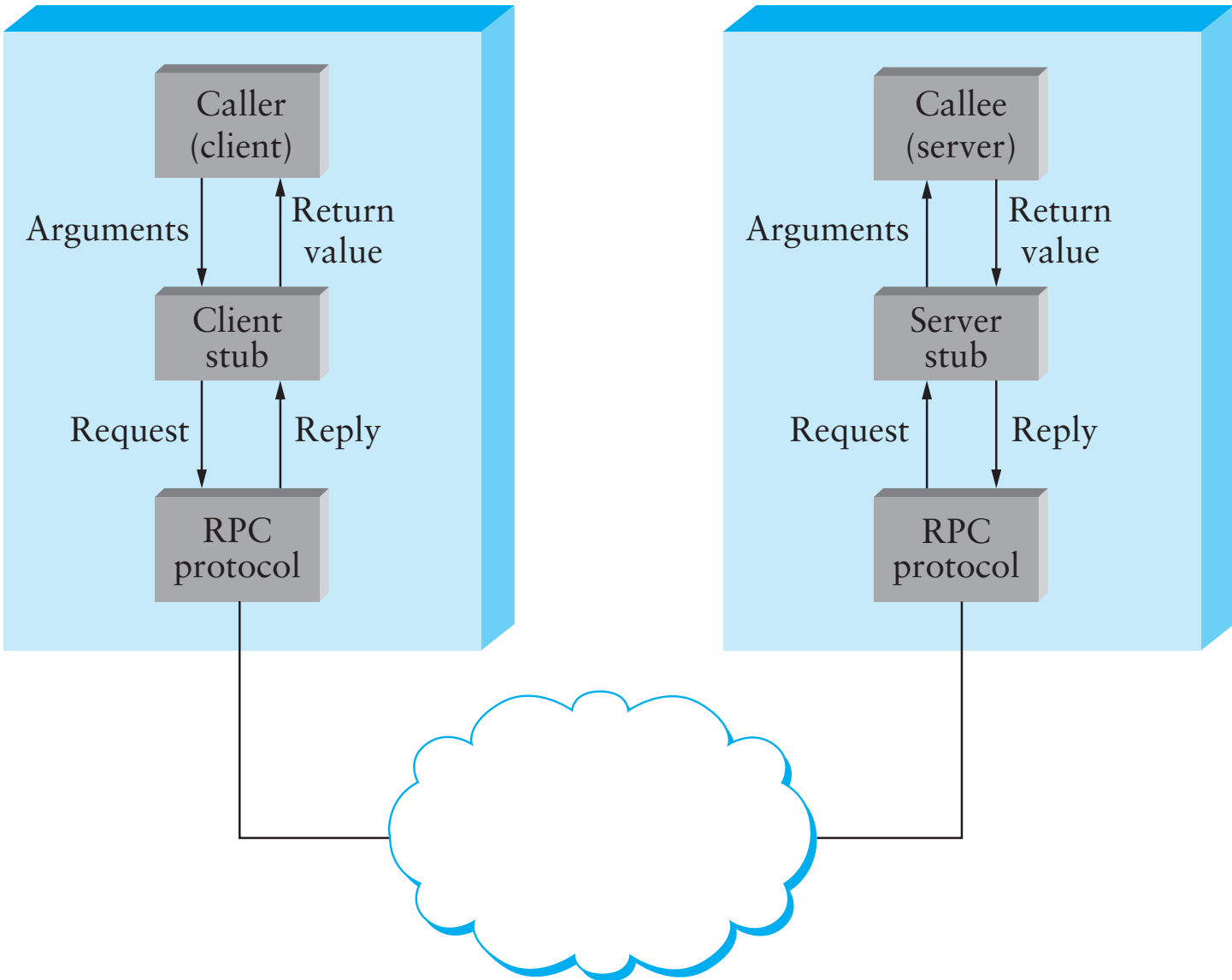
# Can we maintain the same semantics?

- **Mostly…**
- **Why not?**
  - New failure modes: nodes, network
- **Possible outcomes of failure**
  - Procedure did not execute
  - Procedure executed once
  - Procedure executed multiple times
  - Procedure partially executed
- **Desired: at-most-once semantics**

# Implementing at-most-once semantics

- **Problem: request message lost**
  - Client must retransmit requests when it gets no reply

- **Problem: reply message lost**
  - Client may retransmit previously executed request
  - OK if operation is *idempotent*
  - Server must keep "replay cache" to reply to already executed requests

- **Problem: server takes too long executing**
  - Client will retransmit request already in progress
  - Server must recognize duplicate – could reply "in progress"

# Server Crashes

- **Problem: server crashes and reply lost**
  - Can make replay cache persistent – slow
  - Can hope reboot takes long enough for all clients to fail
- **Problem: server crashes during execution**
  - Can log enough to restart partial execution – slow and hard
  - Can hope reboot takes long enough for all clients to fail
- **Can use "cookies" to inform clients of crashes**
  - Server gives client cookie, which is f(time of boot)
  - Client includes cookie with RPC
  - After server crash, server will reject invalid cookie

# RPC Components

- **Stub Compiler**
  - Creates stub methods
  - Creates functions for marshalling and unmarshalling
- **Dispatcher**
  - Demultiplexes programs running on a machine
  - Calls the stub server function
- **Protocol**
  - At-most-once semantics (or not)
  - Reliability, replay caching, version matching
  - Fragmentation, Framing (depending on underlying protocols)

# Examples of RPC Systems

- **SunRPC (now ONC RPC)**
  - The first popular system
  - Used by NSF
  - Not popular for the wide area (security, convenience)
- **Java RMI**
  - Popular with Java
  - Only works among JVMs
- **DCE**
  - Used in ActiveX and DCOM, CORBA
  - Stronger semantics than SunRPC, much more complex

# …even more examples

- **XML-RPC, SOAP**
- **Json-RPC**
- **Apache Thrift**

# Presentation Formatting

- **How to represent data?**

- **Several questions:**
  - Which data types do you want to support?
    - Base types, Flat types, Complex types
  - How to encode data into the wire
  - How to decode the data?
    - Self-describing (tags)
    - Implicit description (the ends *know*)

- **Several answers:**
  - Many frameworks do these things automatically

# Which data types?

- **Basic types**
  - Integers, floating point, characters
  - Some issues: endianness (ntohs, htons), character encoding, IEEE 754

- **Flat types**
  - Strings, structures, arrays
  - Some issues: packing of structures, order, variable length

- **Complex types**
  - Pointers! Must flatten, or serialize data structures

Application data structure

Argument marshaller
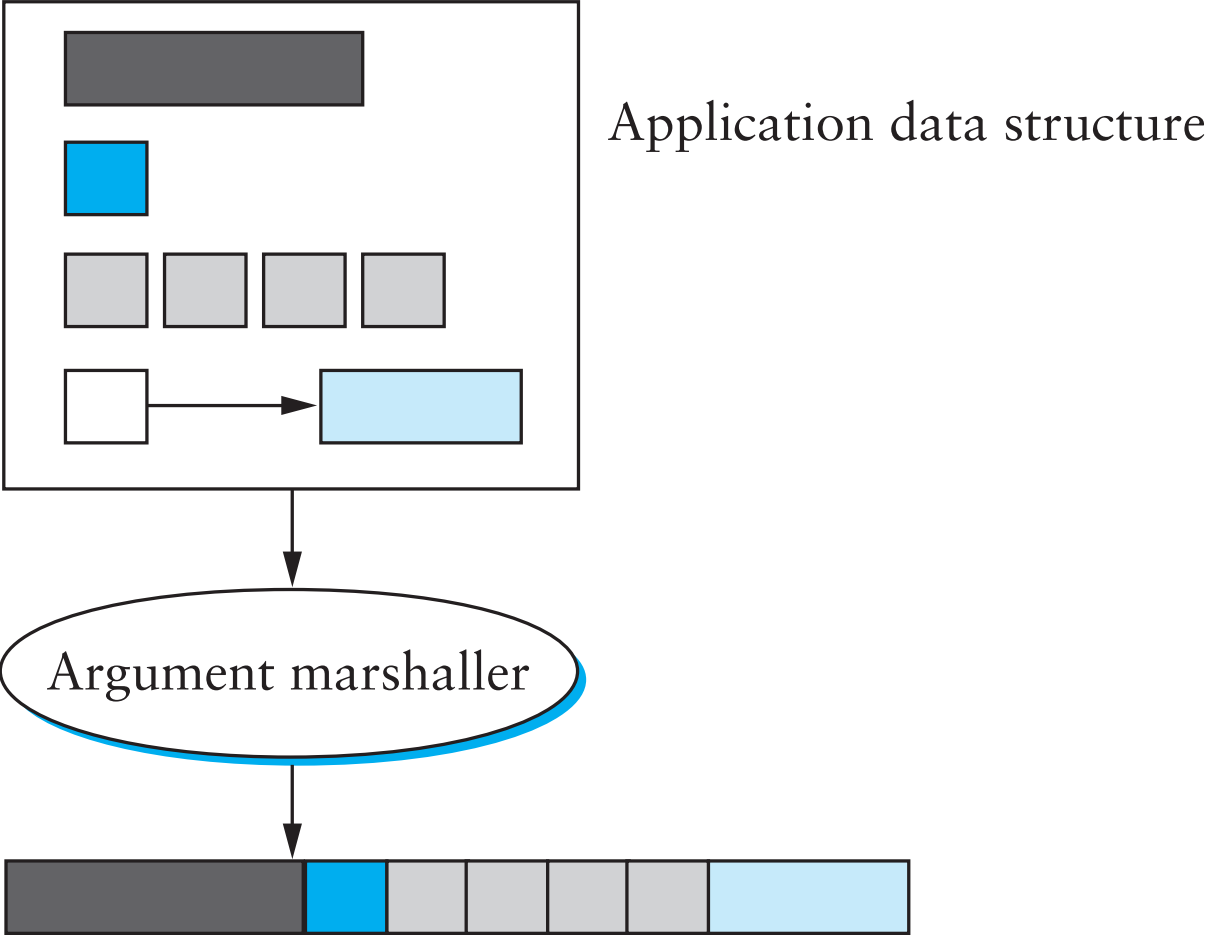
# Data Schema

- **How to parse the encoded data?**

- **Two Extremes:**
  - Self-describing data: tags
    - Additional information added to message to help in decoding
    - Examples: field name, type, length
  - Implicit: the code at both ends "knows" how to decode the message
    - E.g., your Snowcast implementation
    - Interoperability depends on well defined protocol specification!
    - very difficult to change

# Stub Generation

- **Many systems generate stub code from independent specification: IDL**
  - IDL – Interface Description Language
    - describes an interface in a **language neutral** way
- **Separates logical description of data from**
  - Dispatching code
  - Marshalling/unmarshalling code
  - Data wire format

# Example: Sun XDR (RFC 4506)

- *External Data Representation* for SunRPC
- **Types: most of C types**
- **No tags (except for array lengths)**
  - Code needs to know structure of message
- **Usage:**
  - Create a program description file (.x)
  - Run rpcgen program
  - Include generated .h files, use stub functions
- **Very C/C++ oriented**
  - Although encoders/decoders exist for other languages

# Example: fetch and add server

- **In fadd_prot.x:**

```
struct fadd_arg {
  string var<>;
  int inc;
};

union fadd_res switch (bool error) {
case TRUE:
  int sum;
case FALSE:
  string msg<>;
};
```

# RPC Program Definition

```
program FADD_PROG {
  version FADD_VERS {
    void FADDPROC_NULL (void) = 0;
    fadd_res FADDPROC_FADD (fadd_arg) = 1;
  } = 1;
} = 300001;
```

- **Rpcgen generates marshalling/unmarshalling code, stub functions, you fill out the actual code**

# XML

- **Other extreme**

- **Markup language**
  - Text based, semi-human readable
  - Heavily tagged (field names)
  - Depends on external schema for parsing
  - Hard to parse efficiently

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

# Google Protocol Buffers

- **Defined by Google, released to the public**
  - Widely used internally and externally
  - Supports common types, service definitions
  - Natively generates C++/Java/Python code
    - Over 20 other supported by third parties
  - Not a full RPC system, only does marshalling
    - Many third party RPC implementations
  - Efficient binary encoding, readable text encoding
- **Performance**
  - 3 to 10 times smaller than XML
  - 20 to 100 times faster to process

# Protocol Buffers Example

```
message Student {
    required String name = 1;
    required int32 credits = 2;
}
```

**(…compile with proto)**

```
Student s;
s.set_name("Jane");
s.set_credits(20);
fstream output("students.txt" , ios:out | ios:binary );
s.SerializeToOstream(&output);
```

**(…somebody else reading the file)**

```
Student s;
fstream input("students.txt" , ios:in | ios:binary );
s.ParseFromIstream();
```

# Binary Encoding

- **Integers: varints**
  - 7 bits out of 8 to encode integers
  - Msb: more bits to come
  - Multi-byte integers: least significant group first
- **Signed integers: zig-zag encoding, then varint**
  - 0:0, -1:1, 1:2, -2:3, 2:4, …
  - Advantage: smaller when encoded with varint
- **General:**
  - Field number, field type (tag), value
- **Strings:**
  - Varint length, unicode representation

# Apache Thrift

- **Originally developed by Facebook**
- **Used heavily internally**
- **Full RPC system**
  - Support for C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk, and Ocaml
- **Many types**
  - Base types, list, set, map, exceptions
- **Versioning support**
- **Many encodings (protocols) supported**
  - Efficient binary, json encodings

# Apache Avro

- **Yet another newcomer**
- **Likely to be used for Hadoop data representation**
- **Encoding:**
  - Compact binary with schema included in file
  - Amortized self-descriptive
- **Why not just create a new encoding for Thrift?**
  - I don't know…

# Conclusions

- **RPC is good way to structure many distributed programs**
  - Have to pay attention to different semantics, though!
- **Data: tradeoff between self-description, portability, and efficiency**
- **Unless you really want to bit pack your protocol, and it won't change much, use one of the IDLs**
- **Parsing code is easy to get (slightly) wrong, hard to get fast**
  - Should only do this once, for all protocols
- **Which one should you use?**