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
The network between the calling process and the called process has much more complex properties than the backplane of a computer. For example, it is likely to limit message sizes and has a tendency to lose and reorder messages. The computers on which the calling and called processes run may have significantly different architectures and data representation formats. Thus, a complete RPC mechanism actually involves two major components:

1. A protocol that manages the messages sent between the client and the server processes and that deals with the potentially undesirable properties of the underlying network.
2. Programming language and compiler support to package the arguments into a request message on the client machine and then to translate this message back into the arguments on the server machine, and likewise with the return value (this piece of the RPC mechanism is usually called a stub compiler).

Figure 5.12 schematically depicts what happens when a client invokes a remote procedure. First, the client calls a local stub for the procedure, passing it the arguments required by the procedure. This stub hides the fact that the procedure is remote by

<table>
<thead>
<tr>
<th>Caller (client)</th>
<th>Client stub</th>
<th>RPC protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Request</td>
<td>Reply</td>
</tr>
<tr>
<td>Return value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Callee (server)</th>
<th>Server stub</th>
<th>RPC protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td>Request</td>
<td>Reply</td>
</tr>
<tr>
<td>Return value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 $f(\text{time of boot})$
  – 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 NFS
  - 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

- Apache Thrift
- Google gRPC
- XML-RPC, SOAP
- Json-RPC
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, type-length-value)
    • 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
At the next level are flat types—structures and arrays. While flat types might at first not appear to complicate argument marshalling, the reality is that they do. The problem is that the compilers used to compile application programs sometimes insert padding between the fields that make up the structure so as to align these fields on word boundaries. The marshalling system typically packs structures so that they contain no padding.

At the highest level, the marshalling system might have to deal with complex types—those types that are built using pointers. That is, the data structure that one program wants to send to another might not be contained in a single structure, but might instead involve pointers from one structure to another. A tree is a good example of a complex type that involves pointers. Clearly, the data encoder must prepare the data structure for transmission over the network because pointers are implemented by memory addresses, and just because a structure lives at a certain memory address on one machine does not mean it will live at the same address on another machine. In other words, the marshalling system must serialize (flatten) complex data structures.

In summary, depending on how complicated the type system is, the task of argument marshalling usually involves converting the base types, packing the structures, and linearizing the complex data structures, all to form a contiguous message that can be transmitted over the network. Figure 7.3 illustrates this task.

Conversion Strategy

Once the type system is established, the next issue is what conversion strategy the argument marshaller will use. There are two general options: canonical intermediate form and receiver-makes-right. We consider each, in turn.

Figure 7.3 Argument marshalling: converting, packing, and linearizing.
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

• 2 Main ideas:
• Introspection-based
  – E.g., Java RMI
• 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: gRPC

- **IDL-based, defined by Google**
  - Protocol Buffers as IDL
- **User specifies services, calls**
  - Unary and streaming calls
  - Synchronous and Asynchronous
  - Timeouts
  - Cancellations

```protobuf
service HelloService {
    rpc SayHello (HelloRequest) returns (HelloResponse);
}
message HelloRequest {
    string greeting = 1;
}
message HelloResponse {
    string reply = 1;
}
```
gRPC

• Generates stubs in many languages
  – C/C++, C#, Node.js, PHP, Ruby, Python, Go, Java
  – These are interoperable

• Transport is http/2
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
  – Efficient binary encoding, readable text encoding

• Performance
  – 3 to 10 times smaller than XML
  – 20 to 100 times faster to process
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?**