COMPILER-ASSISTED PROTOCOL PROCESSING + REMOTE PROCEDURE CALL (RPC) + GRPC

George Porter
Feb 7 and 14, 2023
ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
REQUIRED READING

Chapter 4 of “Network Programming with Go”
Outline

1. RPC fundamentals
2. Compiler-assisted framing and parsing
3. gRPC demo
WHY RPC?

• The typical programmer is trained to write single-threaded code that runs in one place
• **Goal**: Easy-to-program network communication that makes client-server communication transparent
• Retains the “feel” of writing centralized code
  • Programmer needn’t think about the network
REMOTE PROCEDURE CALL (RPC)

• Distributed programming is challenging
  • Need common primitives/abstraction to hide complexity
  • E.g., file system abstraction to hide block layout, process abstraction for scheduling/fault isolation
• In early 1980’s, researchers at PARC noticed most distributed programming took form of remote procedure call
WHAT’S THE GOAL OF RPC?

• Within a single program, running in a single process, recall the well-known notion of a procedure call:
  • Caller pushes arguments onto stack,
    • jumps to address of callee function
  • Callee reads arguments from stack,
    • executes, puts return value in register,
    • returns to next instruction in caller

RPC’s Goal: To make communication appear like a local procedure call: transparency for procedure calls
RPC EXAMPLE

Local computing

\[ X = 3 \times 10; \]

print(X)

> 30

Remote computing

server = connectToServer(S);

Try:

\[ X = \text{server}.\text{mult}(3,10); \]

print(X)

Except e:

print “Error!”

> 30

or

> Error
RPC ISSUES

- Heterogeneity
  - Client needs to **rendezvous** with the server
  - Server must **dispatch** to the required function
    - What if server is **different** type of machine?
- Failure
  - What if messages get **dropped**?
  - What if client, server, or network **fails**?
- Performance
  - Procedure call takes \( \approx 10 \) cycles \( \approx 3 \) ns
  - RPC in a data center takes \( \approx 10 \mu s \) (\( 10^3 \times \) slower)
    - In the wide area, typically \( 10^6 \times \) slower
PROBLEM: DIFFERENCES IN DATA REPRESENTATION

• Not an issue for local procedure call

• For a remote procedure call, a remote machine may:
  • Represent data types using different sizes
  • Use a different byte ordering (*endianness*)
  • Represent floating point numbers differently
  • Have different data alignment requirements
    • *e.g.*, 4-byte type begins only on 4-byte memory boundary
• x86-64 is a **little endian** architecture
  • **Least** significant byte of multi-byte entity at **lowest** memory address
    • “Little end goes first”
  • Some other systems use **big endian**
    • **Most** significant byte of multi-byte entity at **lowest** memory address
      • “Big end goes first”

```
int 5 at address 0x1000:
0x1000:  0000 0101
0x1001:  0000 0000
0x1002:  0000 0000
0x1003:  0000 0000

int 5 at address 0x1000:
0x1000:  0000 0000
0x1001:  0000 0000
0x1002:  0000 0000
0x1003:  0000 0101
```
PROBLEM: DIFFERENCES IN PROGRAMMING SUPPORT

- Language support varies:

  - Many programming languages have **no inbuilt concept** of remote procedure calls
    - *e.g.*, C, C++, earlier Java

  - Some languages have **support that enables** RPC
    - *e.g.*, Python, Haskell, Go
SOLUTION: INTERFACE DESCRIPTION LANGUAGE

- Mechanism to pass procedure parameters and return values in a **machine-independent way**
- Programmer may write an **interface description** in the IDL
  - Defines API for procedure calls: names, parameter/return types
- Then runs an **IDL compiler** which generates:
  - Code to **marshal** (convert) native data types into machine-independent byte streams
    - And vice-versa, called **unmarshaling**
  - **Client stub:** Forwards local procedure call as a request to server
  - **Server stub:** Dispatches RPC to its implementation
A DAY IN THE LIFE OF AN RPC

1. Client calls stub function (pushes params onto stack)

Client machine

Client process

\[ k = \text{add}(3, 5) \]

Client stub (RPC library)
A DAY IN THE LIFE OF AN RPC

1. Client calls stub function (pushes params onto stack)

2. **Stub marshals parameters to a network message**
A DAY IN THE LIFE OF AN RPC

2. Stub marshals parameters to a network message

3. OS sends a network message to the server
A DAY IN THE LIFE OF AN RPC

3. OS sends a network message to the server

4. Server OS receives message, sends it up to stub

Client machine

Client process
k = add(3, 5)

Client stub (RPC library)

Client OS

Server machine

Server stub (RPC library)

Server OS

proc: add | int: 3 | int: 5
4. Server OS receives message, sends it up to stub

5. **Server stub unmarshals params, calls server function**

Client machine

- Client process
  - `k = add(3, 5)`
- Client stub (RPC library)
- Client OS

Server machine

- Server process
  - Implementation of add
- Server stub (RPC library)
  - `proc: add | int: 3 | int: 5`
- Server OS
5. Server stub unmarshals params, calls server function

6. Server function runs, returns a value
A DAY IN THE LIFE OF AN RPC

6. Server function runs, returns a value

7. Server stub marshals the return value, sends msg
A DAY IN THE LIFE OF AN RPC

7. Server stub marshals the return value, sends msg

8. **Server OS sends the reply back across the network**

Client machine
- Client process $k = \text{add}(3, 5)$
- Client stub (RPC library)
- Client OS

Server machine
- Server process $8 \leftarrow \text{add}(3, 5)$
- Server stub (RPC library)
- Server OS
- Result | int: 8
A DAY IN THE LIFE OF AN RPC

8. Server OS sends the reply back across the network

9. Client OS receives the reply and passes up to stub

Client machine
- Client process: \( k = \text{add}(3, 5) \)
- Client stub (RPC library)
- Client OS
  - Result: int: 8

Server machine
- Server process: \( 8 \leftarrow \text{add}(3, 5) \)
- Server stub (RPC library)
- Server OS
9. Client OS receives the reply and passes up to stub

10. Client stub unmarshals return value, returns to client
PETERSON AND DAVIE VIEW

Diagram showing the interaction between a client and a server using an RPC protocol. The client stub receives arguments and requests a return value, while the server stub receives arguments and returns a value. The diagram also includes a cloud symbol to indicate remote communication.
THE SERVER STUB IS REALLY TWO PARTS

- **Dispatcher**
  - Receives a client’s RPC request
    - **Identifies** appropriate server-side method to invoke
- **Skeleton**
  - **Unmarshals** parameters to server-native types
  - **Calls** the local server procedure
  - **Marshals** the response, sends it back to the dispatcher

- **All this is hidden from the programmer**
  - Dispatcher and skeleton may be integrated
    - Depends on implementation
Outline

1. RPC fundamentals
2. Compiler-assisted framing and parsing
3. gRPC demo
<table>
<thead>
<tr>
<th>fred</th>
<th>programmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>liping</td>
<td>analyst</td>
</tr>
<tr>
<td>sureerat</td>
<td>manager</td>
</tr>
</tbody>
</table>
3
fred
programmer
liping
analyst
sureerat
manager
ANOTHER OPTION

Assumes that the first column is 8 chars wide, and the 2nd is 10 chars wide
TAKE-AWAY

Lots of valid ways of encoding complex data...

...but both endpoints need to know how to interpret what the other side sends them.
GO’S AUTOMATIC ENCODING SUPPORT

```
gmporter@navygrog ch4 % go doc encoding
package encoding // import "encoding"

Package encoding defines interfaces shared by other packages for data in and out of byte-level and textual representations. Packages for these interfaces include encoding/gob, encoding/json, and net.IP. The interfaces come in pairs that produce and consume:

```
GO’S ENCODING FORMATS

gmporter@navygrog ch4 % go list encoding/...
encoding
encoding/ascii85
encoding/asn1
encoding/base32
encoding/base64
encoding/binary
encoding/csv
encoding/gob
encoding/hex
encoding/json
encoding/pem
encoding/xml

gmporter@navygrog ch4 %
(BASE64 DEMO)
A CLOSER LOOK USING JSON

• Javascript Object Notation commonly used in networked applications

• Youtube search, Google Maps, twitter

• Let’s look at a tweet:
  • https://twitter.com/georgemporter/status/1623027391642877953
(TWITTER DEMO)
(JSON ECHO CLIENT AND SERVER DEMO)
PROTOBUF: INTERFACE DEFINITION LANGUAGE

- Language-neutral way of specifying:
  - Data structures called Messages (Protocol Buffers)
  - Services, consisting of procedures/methods (gRPC)
- Stub compiler
  - Compiles IDL into Python, Java, etc. (protoc)
IDL LANGUAGE: PROTOCOL BUFFERS

- Defines Messages (i.e., data structures) language neutral

```python
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

- Field 1: query
  - Type: String

- Field 2: page_number
  - Type: 32-bit signed int

- Field 3: result_per_page
  - Type: 32-bit signed int

We’re using version 3 of protocol buffers

Name of the message
<table>
<thead>
<tr>
<th>.proto Type</th>
<th>Notes</th>
<th>C++ Type</th>
<th>Java/Kotlin Type[^1]</th>
<th>Python Type[^2]</th>
<th>Go Type</th>
<th>Ruby Type</th>
<th>C# Type</th>
<th>PHP Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td></td>
<td>double</td>
<td>double</td>
<td>float</td>
<td>float64</td>
<td>Float</td>
<td>double</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>float</td>
<td>float</td>
<td>float32</td>
<td>float32</td>
<td>Float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>int32</td>
<td>Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.</td>
<td>int32</td>
<td>int</td>
<td>int</td>
<td>int32</td>
<td>Fixnum or Bignum (as required)</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>int64</td>
<td>Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.</td>
<td>int64</td>
<td>long</td>
<td>int/long[^4]</td>
<td>int64</td>
<td>Bignum</td>
<td>long</td>
<td>integer/str</td>
</tr>
<tr>
<td>uint32</td>
<td>Uses variable-length encoding.</td>
<td>uint32</td>
<td>int[^2]</td>
<td>int/long[^4]</td>
<td>uint32</td>
<td>Fixnum or Bignum (as required)</td>
<td>uint</td>
<td>integer</td>
</tr>
<tr>
<td>sint32</td>
<td>Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.</td>
<td>int32</td>
<td>int</td>
<td>int</td>
<td>int32</td>
<td>Fixnum or Bignum (as required)</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>.proto Type</td>
<td>Notes</td>
<td>C++ Type</td>
<td>Java/Kotlin Type</td>
<td>Python Type</td>
<td>Go Type</td>
<td>Ruby Type</td>
<td>C# Type</td>
<td>PHP Type</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>----------</td>
<td>------------------</td>
<td>-------------</td>
<td>---------</td>
<td>----------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>fixed32</td>
<td>Always four bytes. More efficient than uint32 if values are often greater than $2^{28}$.</td>
<td>uint32</td>
<td>int$^2$</td>
<td>int/long$^4$</td>
<td>uint32</td>
<td>Fixnum or Bignum (as required)</td>
<td>uint</td>
<td>integer</td>
</tr>
<tr>
<td>fixed64</td>
<td>Always eight bytes. More efficient than uint64 if values are often greater than $2^{56}$.</td>
<td>uint64</td>
<td>long$^2$</td>
<td>int/long$^4$</td>
<td>uint64</td>
<td>Bignum</td>
<td>ulong</td>
<td>integer/string</td>
</tr>
<tr>
<td>sfixed32</td>
<td>Always four bytes.</td>
<td>int32</td>
<td>int</td>
<td>int</td>
<td>int32</td>
<td>Fixnum or Bignum (as required)</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>sfixed64</td>
<td>Always eight bytes.</td>
<td>int64</td>
<td>long</td>
<td>int/long$^4$</td>
<td>int64</td>
<td>Bignum</td>
<td>long</td>
<td>integer/string</td>
</tr>
<tr>
<td>bool</td>
<td></td>
<td>bool</td>
<td>boolean</td>
<td>bool</td>
<td>bool</td>
<td>TrueClass/FalseClass</td>
<td>bool</td>
<td>boolean</td>
</tr>
<tr>
<td>string</td>
<td>A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than $2^{32}$.</td>
<td>string</td>
<td>String</td>
<td>str/unicode$^5$</td>
<td>string</td>
<td>String (UTF-8)</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>bytes</td>
<td>May contain any arbitrary sequence of bytes no longer than $2^{32}$.</td>
<td>string</td>
<td>ByteString</td>
<td>str (Python 2) bytes (Python 3)</td>
<td>[]byte</td>
<td>String (ASCII-8BIT)</td>
<td>ByteString</td>
<td>string</td>
</tr>
</tbody>
</table>
• Why do we label the fields with numbers?

• So we can change “signature” of the message later and still be compatible with legacy code

```protobuf
syntax = "proto3";
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```
GOOGLE RPC (GRPC)

- Cross-platform RPC toolkit developed by Google
- Languages:
  - C++, Java, Python, Go, Ruby, C#, Node.js, Android, Obj-C, PHP
- Defines services
- Collection of RPC calls

```plaintext
service Search {
  rpc searchWeb(SearchRequest) returns (SearchResult) {}
}
```
MAKING SERVICES *EVOLVABLE*

- No way to “stop everything” and upgrade
- Clients/servers/services must co-exist
- For newly added fields, old services use defaults:
  - **String**: “”
  - **bytes**: []
  - **bools**: false
  - **numeric**: 0
  - …
PROTOCOL BUFFERS: MAP TYPE

- map<key_type, value_type> map_field = N;

Example:
- map<string, Project> projects = 3;
IMPLEMENTING IN DIFFERENT LANGUAGES

IDL

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

C++: reading from a file

```cpp
Person john;
fstream input(argv[1],
             ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

Java: writing to a file

```java
Person john = Person.newBuilder()
  .setId(1234)
  .setName("John Doe")
  .setEmail("jdoe@example.com")
  .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```
A C++ EXAMPLE

```cpp
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.SerializeToOutputStream(&output);

fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromInputStream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

- Can read/write protobuf Message objects to files/stream/raw sockets
- In particular, gRPC service RPCs
  - Take Message as argument, return Message as response
UC San Diego