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

George Porter
Feb 7 and 14, 2023

UC San Diego

ATTRIBUTION

UC San Diego

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 * 10;
print(X)
> 30
```

Remote computing

```
server = connectToServer(S);
Try:
  X = server.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 ≈ 10 cycles ≈ 3 ns
  - RPC in a data center takes ≈ 10 μs (10^3× slower)
    - In the wide area, typically 10^6× 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

BYTE ORDER

- 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

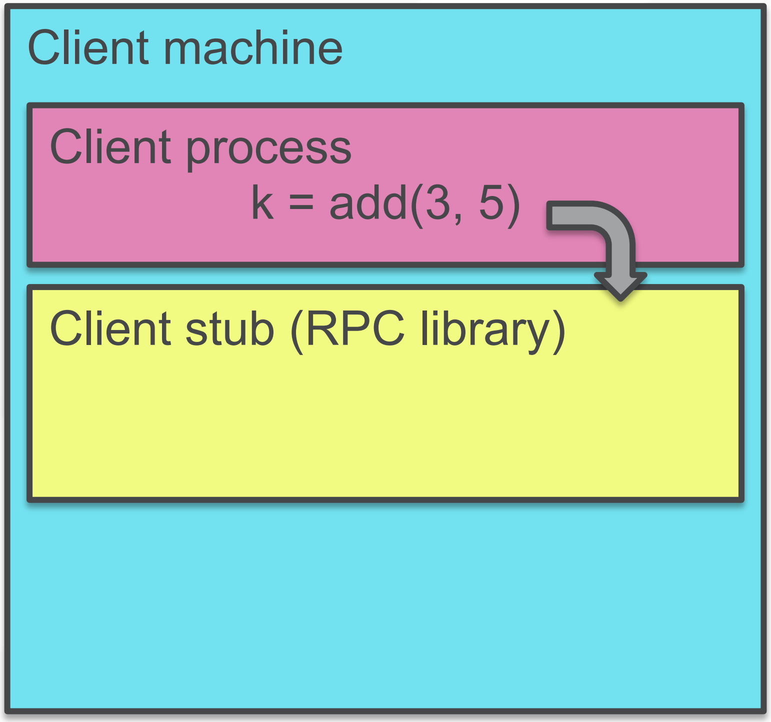
- Language support **varies:**
  - Many programming languages have **no inbuilt concept** of remote procedure calls
    - *e.g.,* C, C++, earlier Java
  - Some languages have **support that enables RPC**
    - *e.g.,* Python, Haskell, Go

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

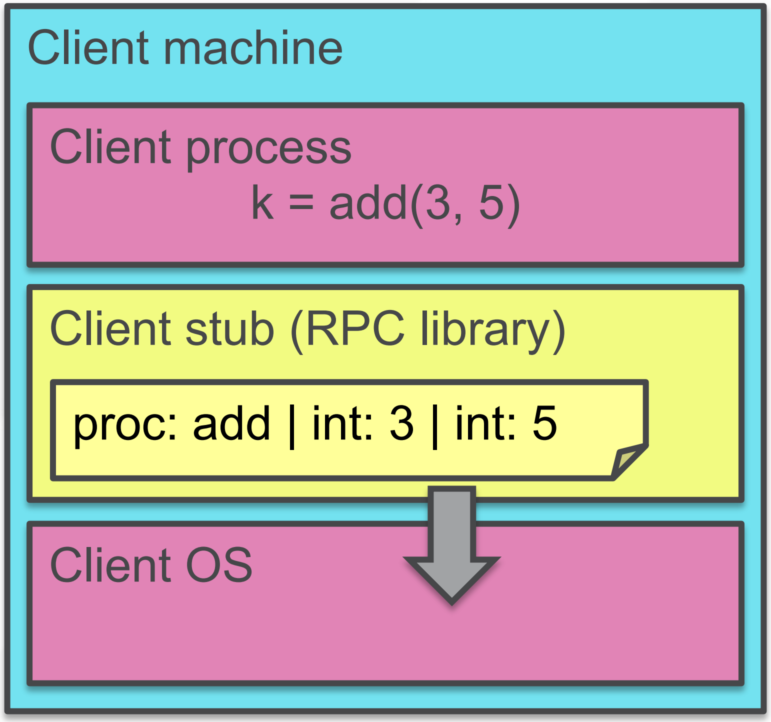
1. Client calls stub function (pushes params onto stack)



A DAY IN THE LIFE OF AN RPC

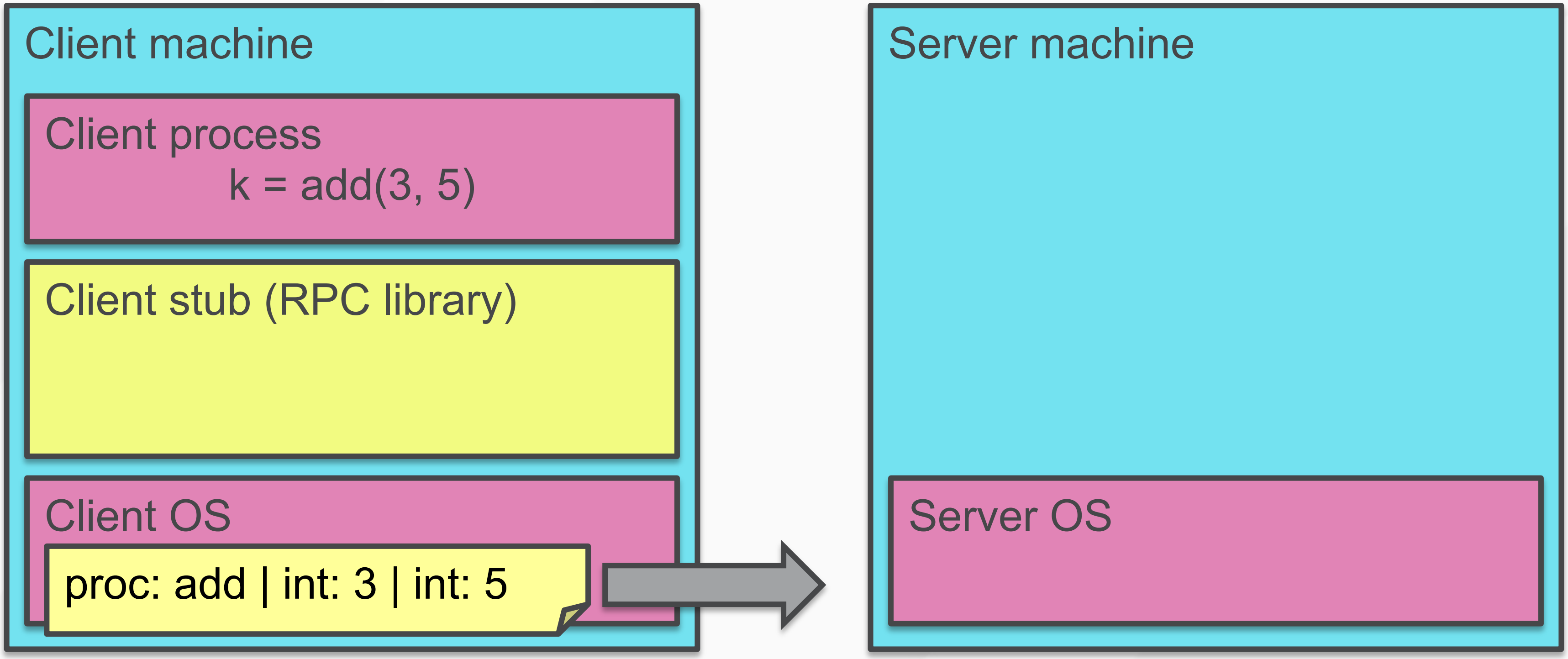
1. Client calls stub function (pushes params onto stack)

2. **Stub marshals parameters to a network message**



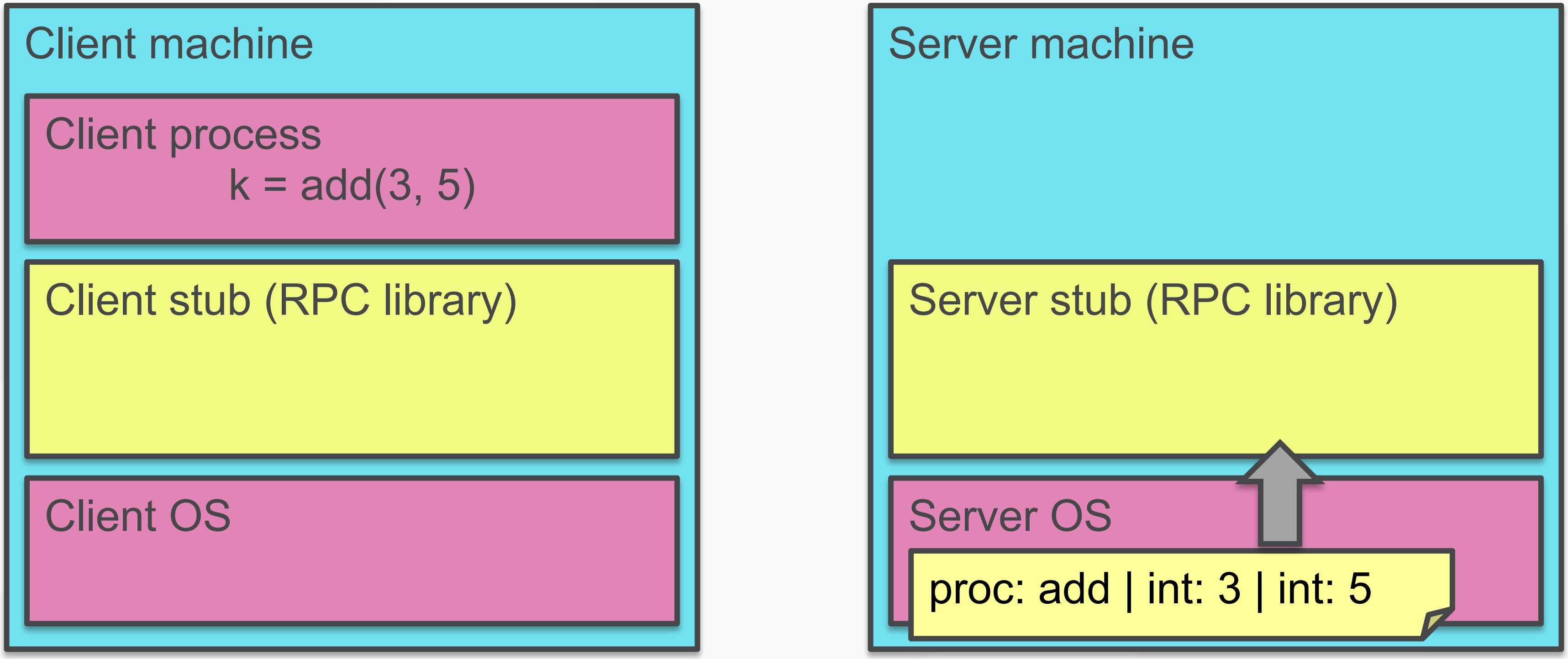
A DAY IN THE LIFE OF AN RPC

2. Stub marshals parameters to a network message
3. **OS sends a network message to the server**



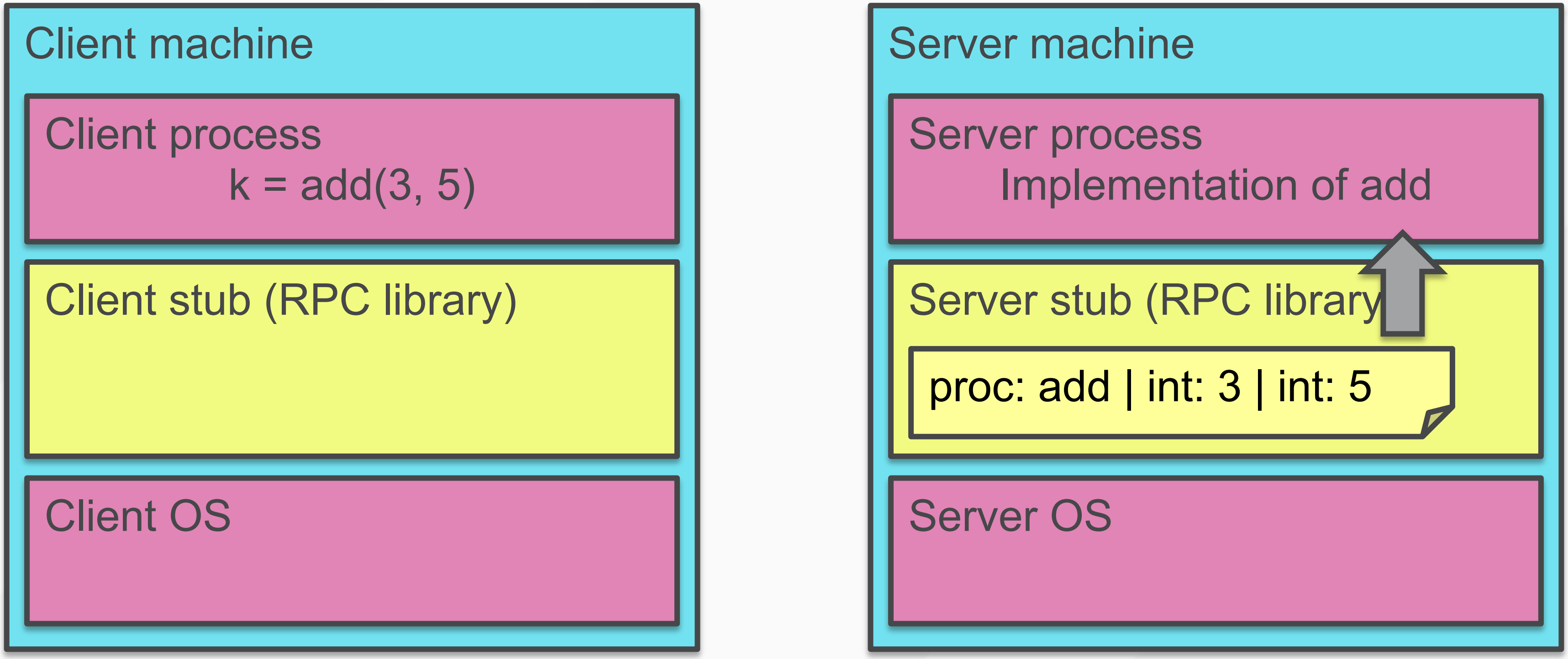
A DAY IN THE LIFE OF AN RPC

3. OS sends a network message to the server
4. **Server OS receives message, sends it up to stub**



A DAY IN THE LIFE OF AN RPC

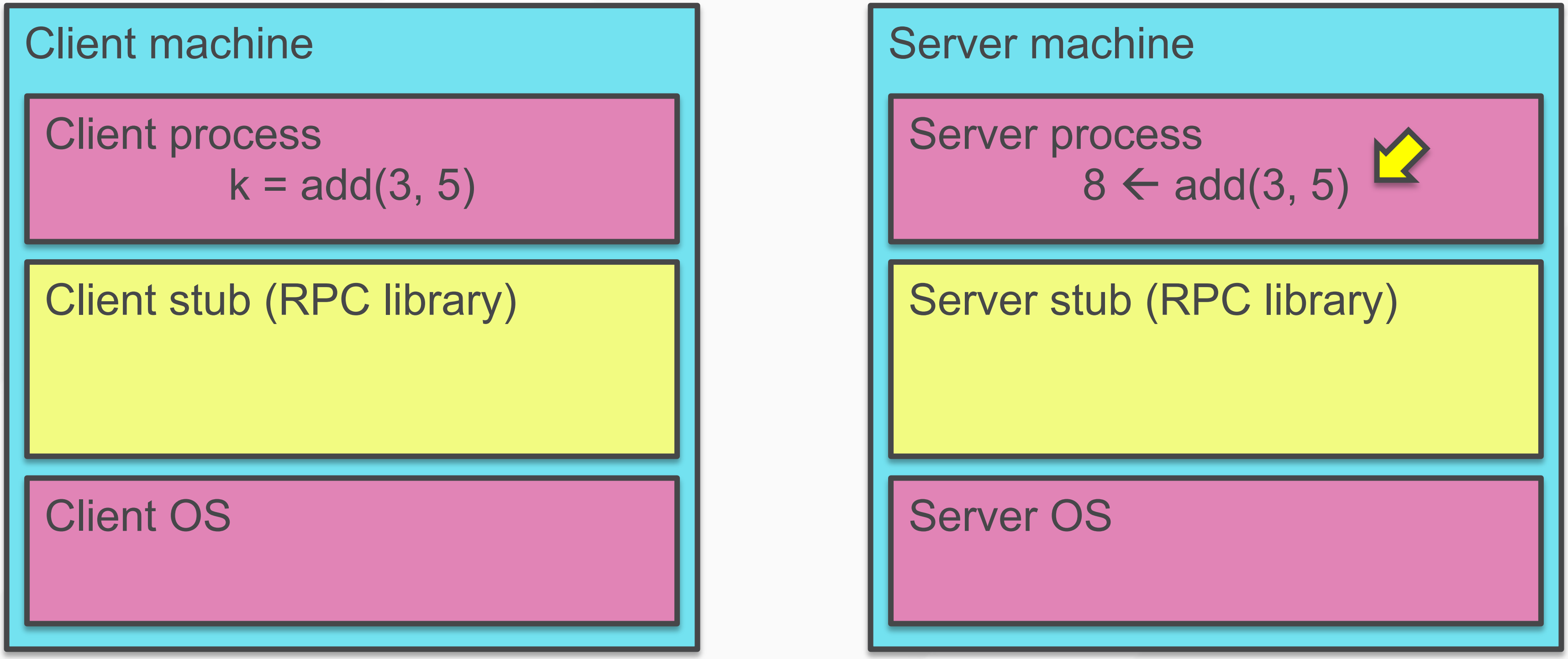
4. Server OS receives message, sends it up to stub
5. **Server stub unmarshals params, calls server function**



A DAY IN THE LIFE OF AN RPC

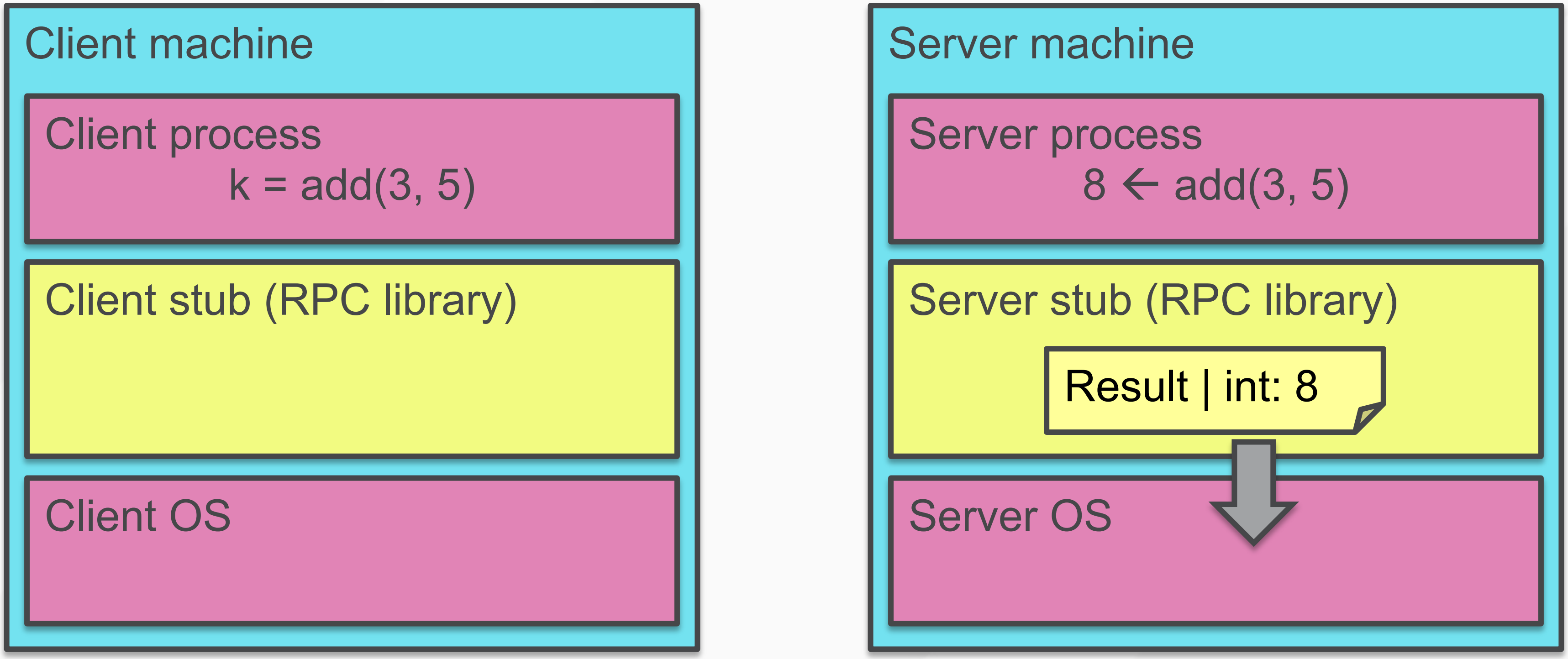
5. Server stub unmarshals params, calls server function

6. **Server function runs, returns a value**



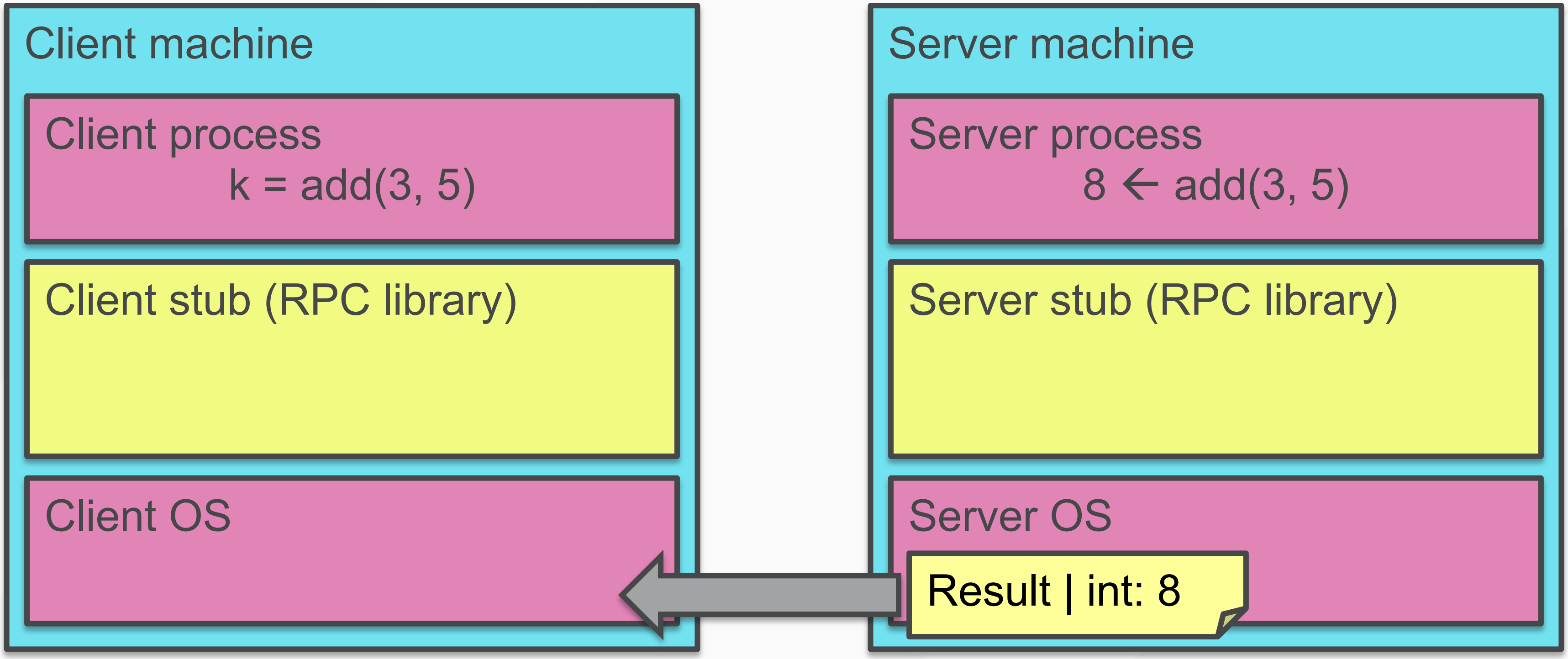
A DAY IN THE LIFE OF AN RPC

6. Server function runs, returns a value
7. **Server stub marshals the return value, sends msg**



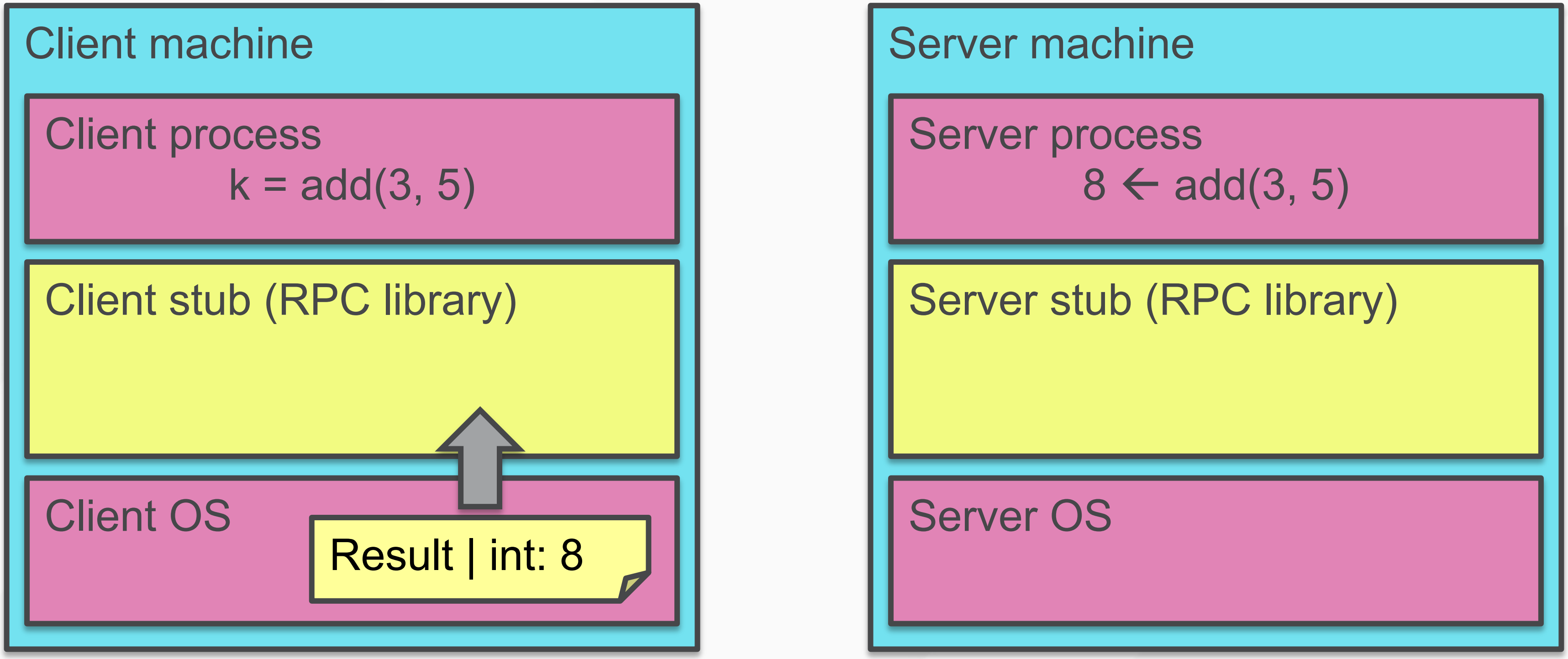
A DAY IN THE LIFE OF AN RPC

7. Server stub marshals the return value, sends msg
8. **Server OS sends the reply back across the network**



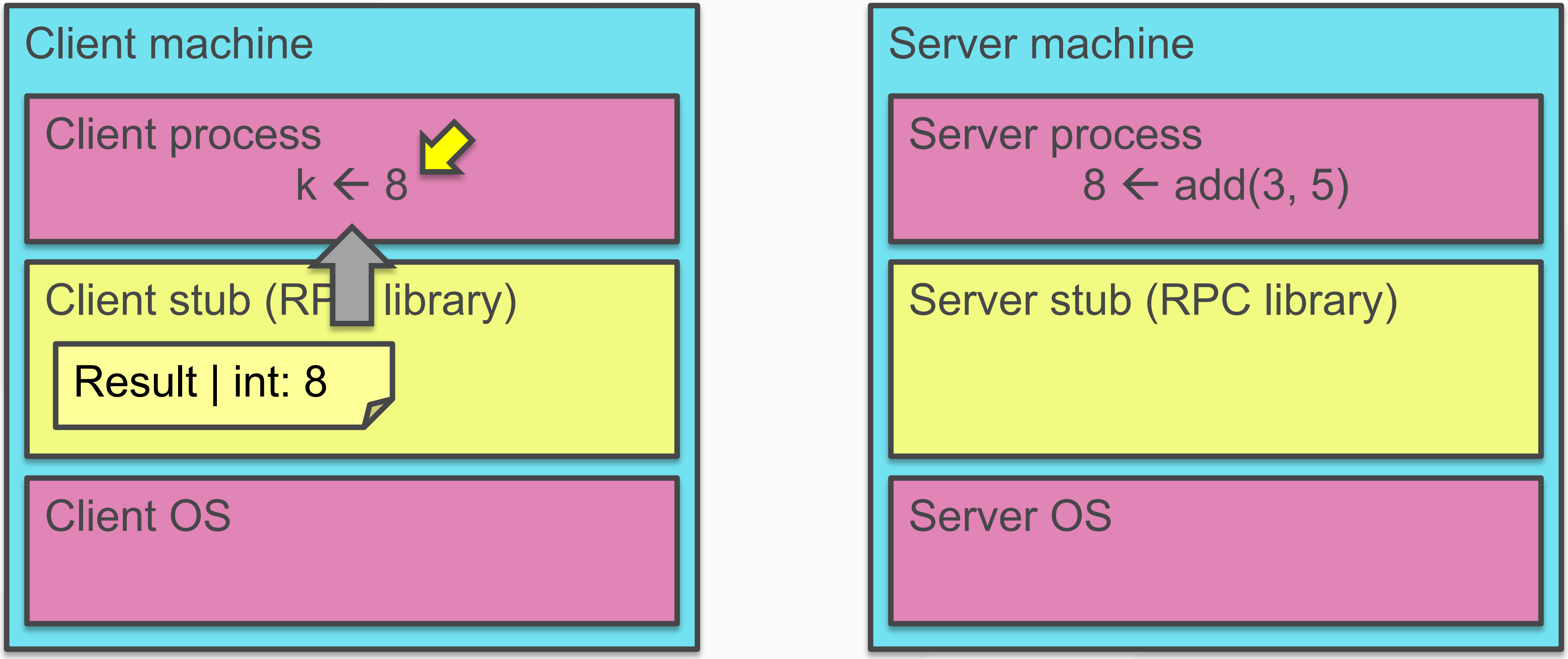
A DAY IN THE LIFE OF AN RPC

8. Server OS sends the reply back across the network
9. **Client OS receives the reply and passes up to stub**

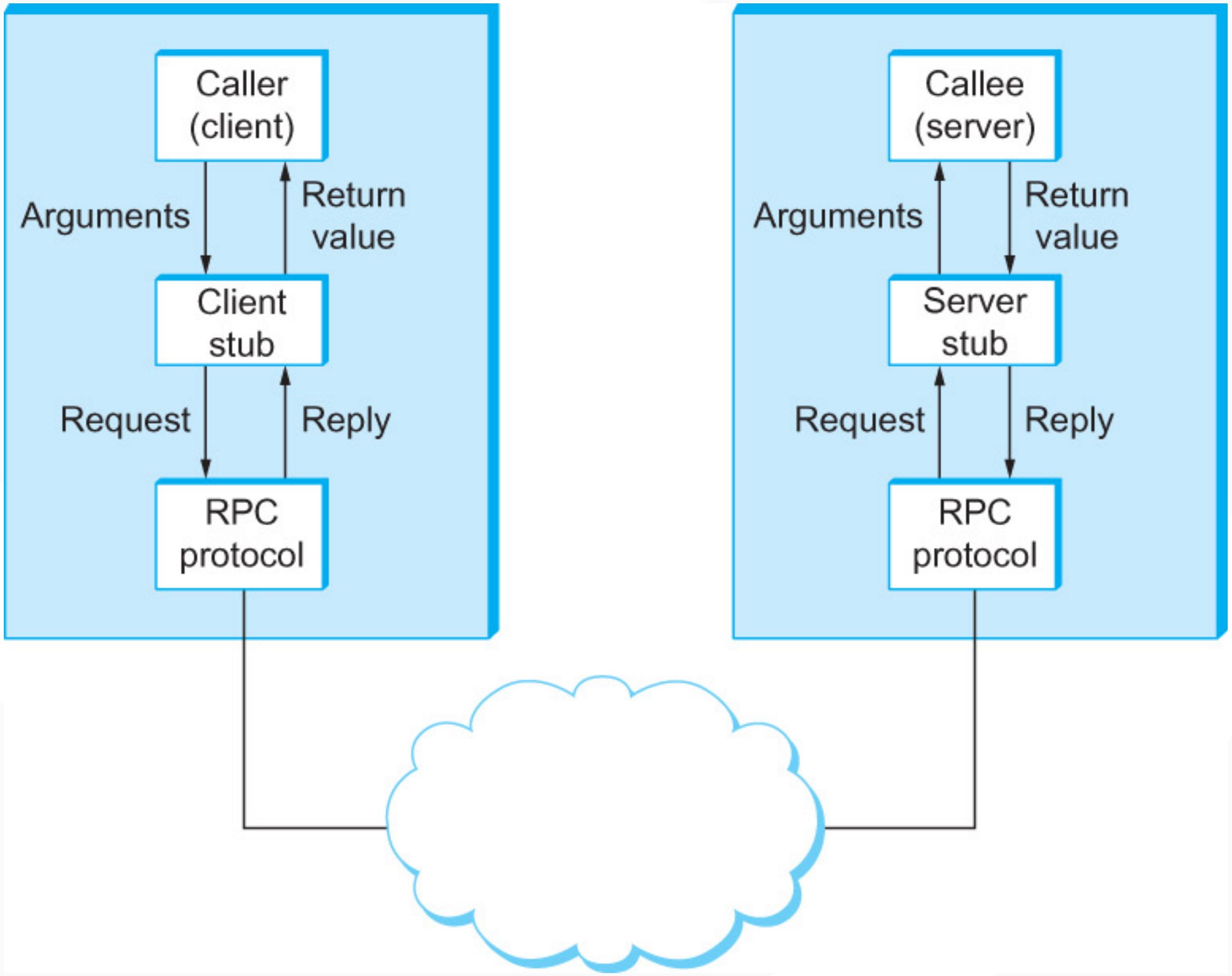


A DAY IN THE LIFE OF AN RPC

9. Client OS receives the reply and passes up to stub
10. **Client stub unmarshals return value, returns to client**



PETERSON AND DAVIE VIEW



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

HOW DO YOU ENCODE THIS TABLE?

fred	programmer
liping	analyst
sureerat	manager

ONE OPTION

```
3
fred\0
programmer\0
liping\0
analyst\0
sureerat\0
manager\0
```

ANOTHER OPTION

```
fred\0\0\0\0
programmer
liping\0\0
analyst\0\0\0
sureerat
manager\0\0\0
```

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
data to and from byte-level and textual representations. Pac
for these interfaces include encoding/gob, encoding/json, an
As a result, implementing an interface once can make a type
encodings. Standard types that implement these interfaces in
net.IP. The interfaces come in pairs that produce and consum

type BinaryMarshaler interface{ ... }
type BinaryUnmarshaler interface{ ... }
type TextMarshaler interface{ ... }
type TextUnmarshaler interface{ ... }
```

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)

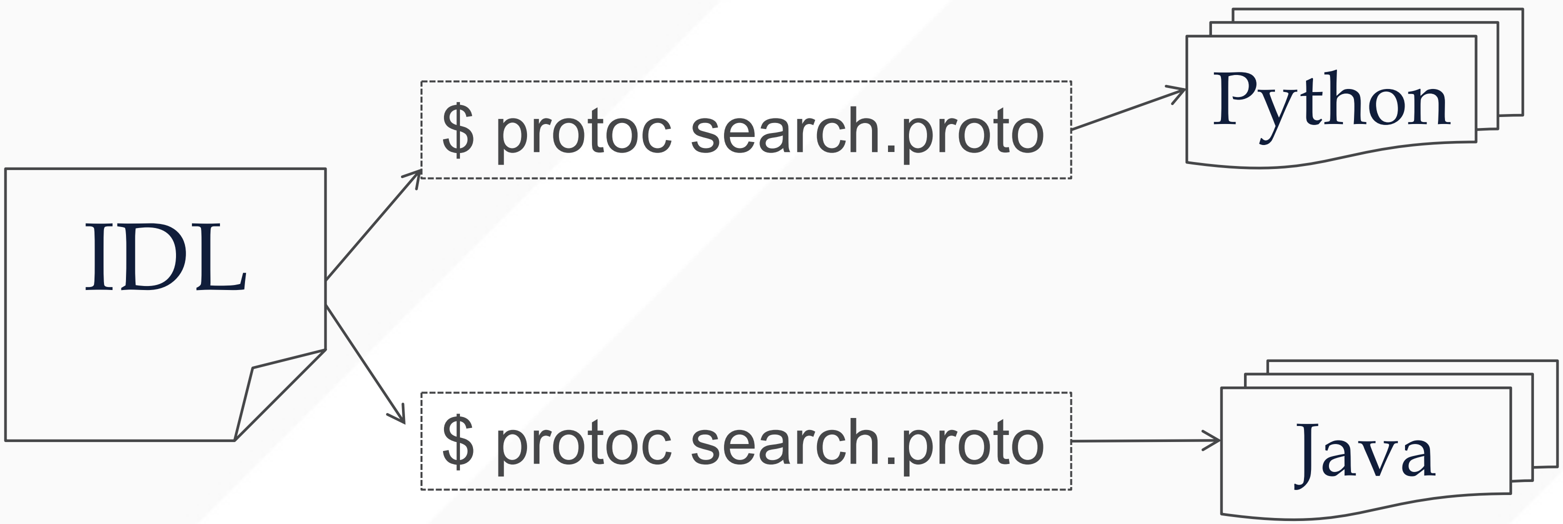
A CLOSER LOOK USING JSON

- 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

We're using version 3 of protocol buffers

Name of the message

Field 1: query
Type: String

Field 2:
page_number
Type: 32-bit
signed int

Field 3:
results_per_page
Type: 32-bit
signed int

```
syntax = "proto3";

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

IDL TYPES MAPPED TO SPECIFIC LANGUAGE TYPES

.proto Type	Notes	C++ Type	Java/Kotlin Type[1]	Python Type[3]	Go Type	Ruby Type	C# Type	PHP Type
double		double	double	float	float64	Float	double	float
float		float	float	float	float32	Float	float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long[4]	int64	Bignum	long	integer/strin
uint32	Uses variable-length encoding.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)	uint	integer
uint64	Uses variable-length encoding.	uint64	long[2]	int/long[4]	uint64	Bignum	ulong	integer/strin
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer

IDL TYPES MAPPED TO SPECIFIC LANGUAGE TYPES

.proto Type	Notes	C++ Type	Java/Kotlin Type[1]	Python Type[3]	Go Type	Ruby Type	C# Type	PHP Type
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int[2]	int/long[4]	uint32	Fixnum or Bignum (as required)	uint	integer
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long[2]	int/long[4]	uint64	Bignum	ulong	integer/strin
sfixed32	Always four bytes.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer
sfixed64	Always eight bytes.	int64	long	int/long[4]	int64	Bignum	long	integer/strin
bool		bool	boolean	bool	bool	TrueClass/FalseClass	bool	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	string	String	str/unicode[5]	string	String (UTF-8)	string	string
bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	string	ByteString	str (Python 2) bytes (Python 3)	[]byte	String (ASCII-8BIT)	ByteString	string

IDL POSITIONAL ARGUMENTS

- Why do we label the fields with numbers?
- So we can change "signature" of the message later and still be compatible with legacy code

```
syntax = "proto3";

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

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 shard_num = 4;
}
```

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

```
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

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

C++: reading from a file

```
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

```
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

```
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);
```

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&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