RPCS AND GRPC

George Porter
Apr 24, 2018

UC San Diego

ATTRIBUTION

UC San Diego

Outline

1. RPC fundamentals
2. Handling failures in RPCs
3. gRPC: Google RPC overview
4. ATM Server 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

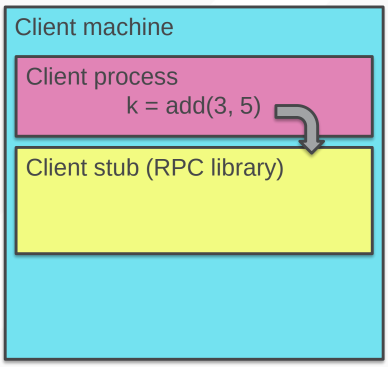
- Language support **varies:**
  - Many programming languages have **no inbuilt concept** of remote procedure calls
    - *e.g.,* C, C++, earlier Java
  - Some languages have **support that enables RPC**
    - *e.g.,* Python, Haskell, Go

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

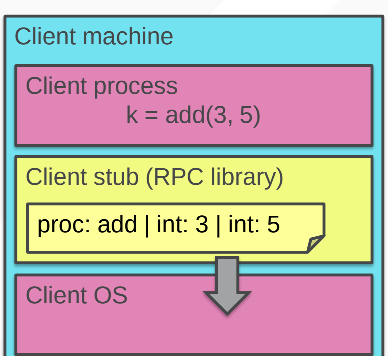
A DAY IN THE LIFE OF AN RPC

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



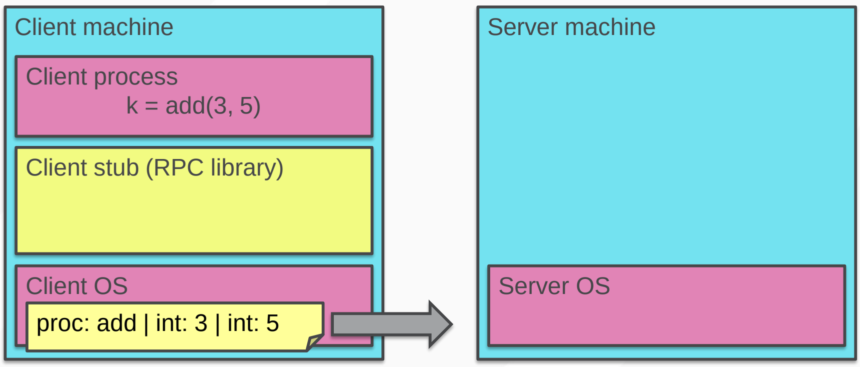
A DAY IN THE LIFE OF AN RPC

1. Client calls stub function (pushes params onto stack)
2. **Stub marshals parameters to a network message**



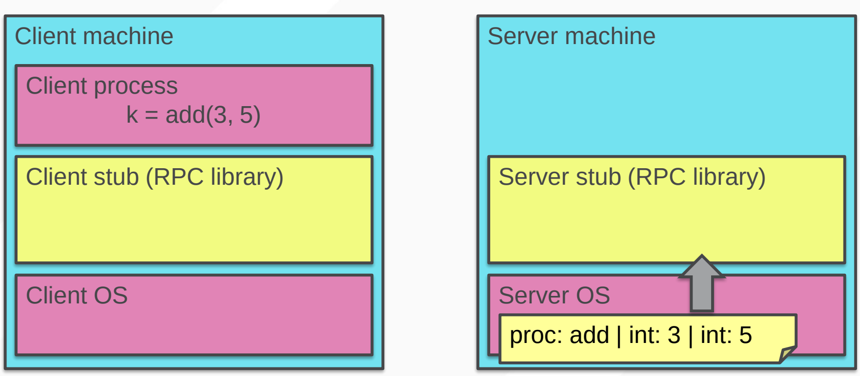
A DAY IN THE LIFE OF AN RPC

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



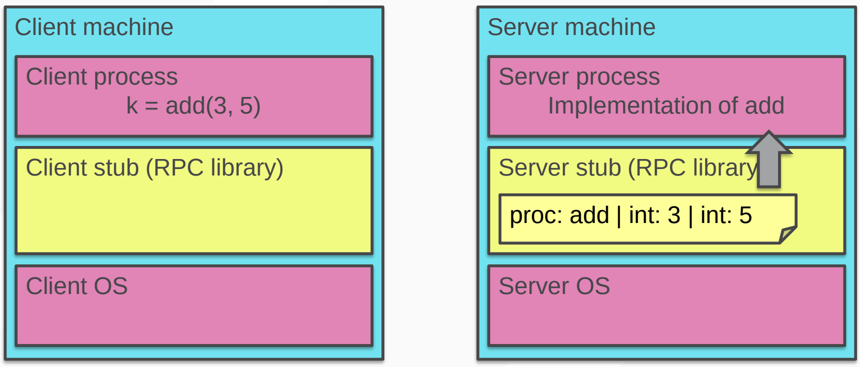
A DAY IN THE LIFE OF AN RPC

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



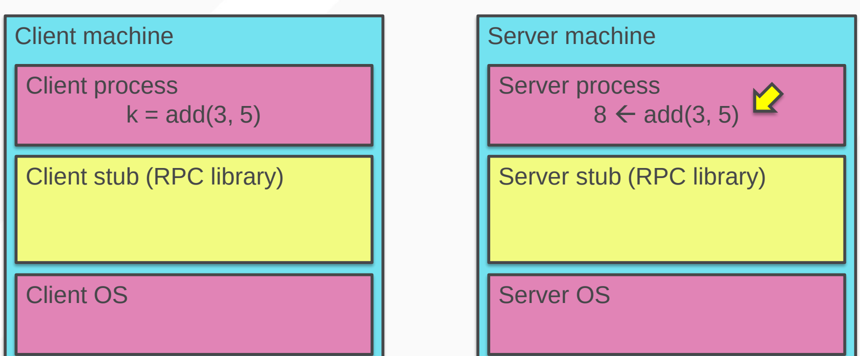
A DAY IN THE LIFE OF AN RPC

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



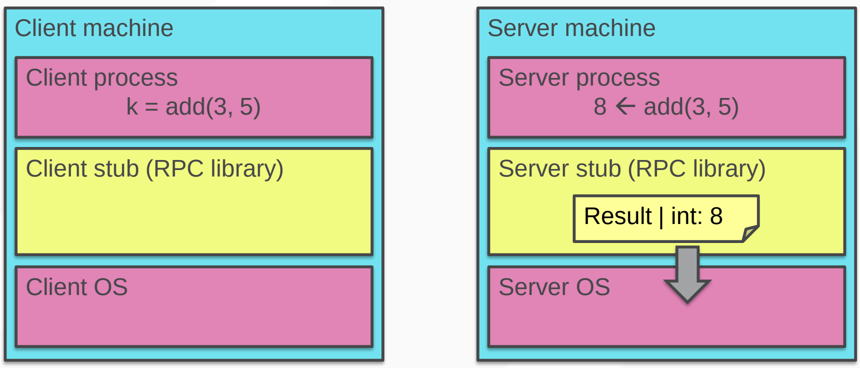
A DAY IN THE LIFE OF AN RPC

5. Server stub unmarshals params, calls server function
6. **Server function runs, returns a value**



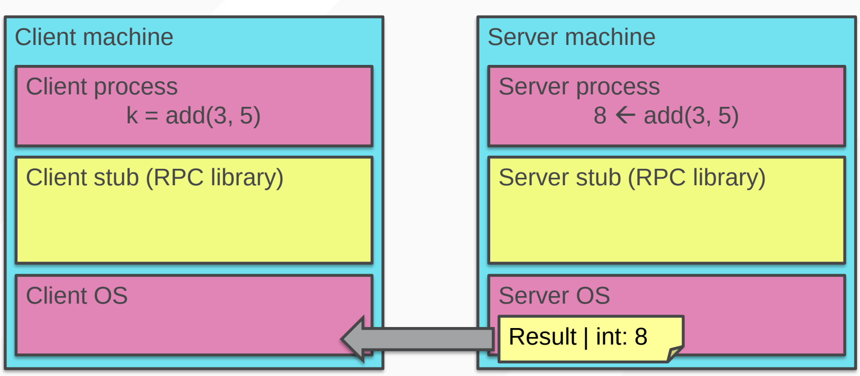
A DAY IN THE LIFE OF AN RPC

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



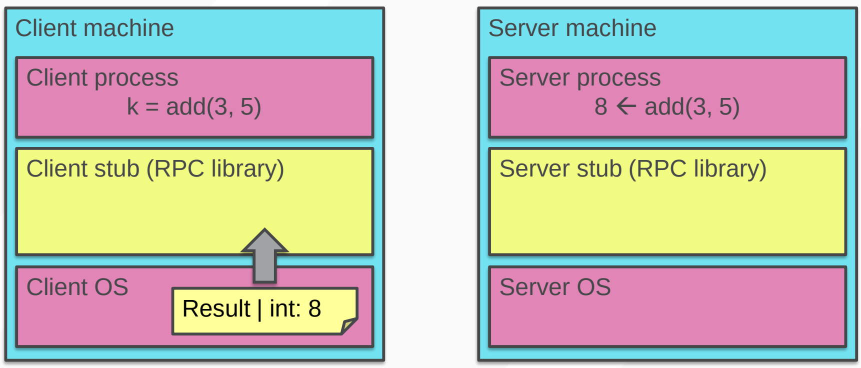
A DAY IN THE LIFE OF AN RPC

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



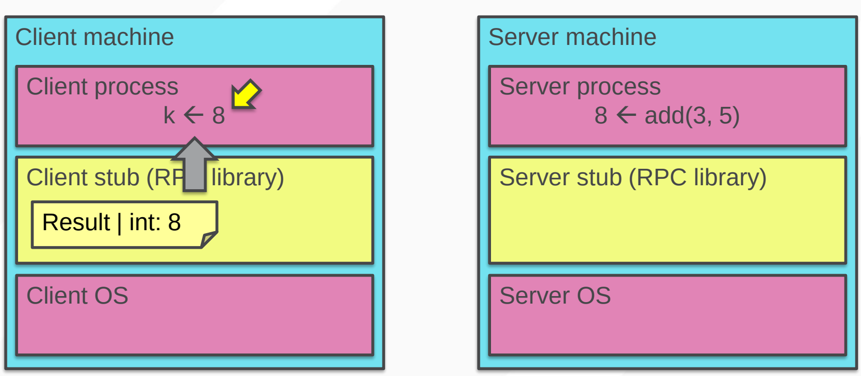
A DAY IN THE LIFE OF AN RPC

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

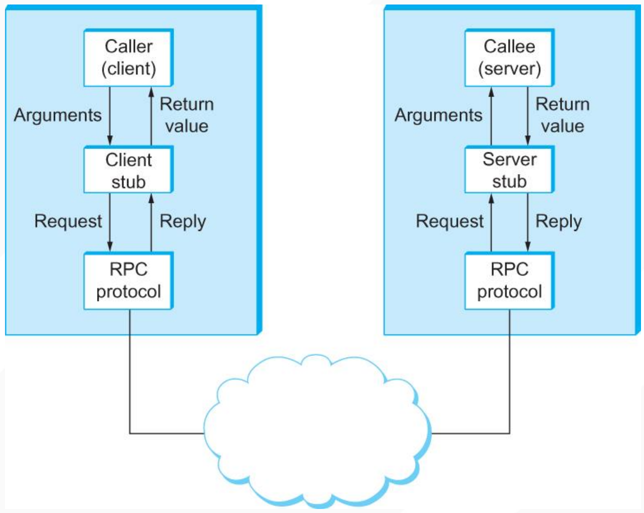


A DAY IN THE LIFE OF AN RPC

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. Handling failures in RPCs
3. gRPC: Google RPC overview
4. ATM Server demo

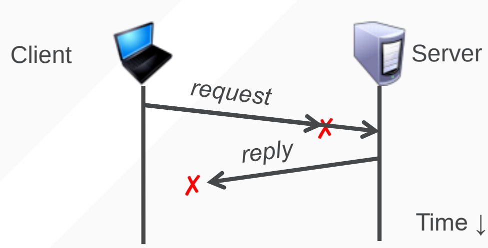
WHAT COULD POSSIBLY GO WRONG?

WHAT COULD POSSIBLY GO WRONG?

1. Client may **crash and reboot**
2. Packets may be **dropped**
   - Some individual **packet loss** in the Internet
   - **Broken routing** results in many lost packets
3. Server may **crash and reboot**
4. Network or server might just be **very slow**

All these may look the same to the client…

FAILURES, FROM CLIENT'S PERSPECTIVE



The cause of the failure is hidden from the client!

AT-LEAST-ONCE SCHEME

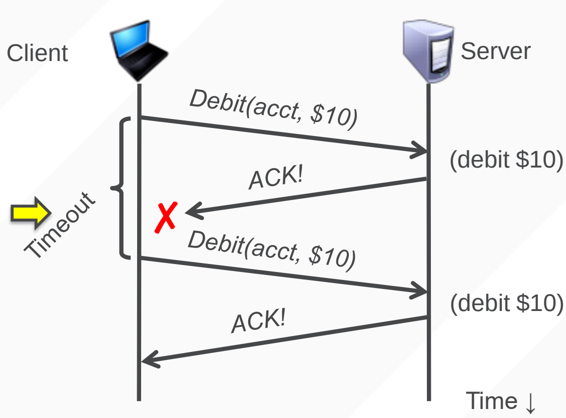
- **Simplest** scheme for handling failures

1. Client stub **waits for a response**, for a while
   - Response takes the form of an ***acknowledgement*** message from the server stub
2. If no response arrives after a fixed ***timeout*** time period, then client stub **re-sends the request**

- Repeat the above a few times
  - *Still no response?* Return an error to the application

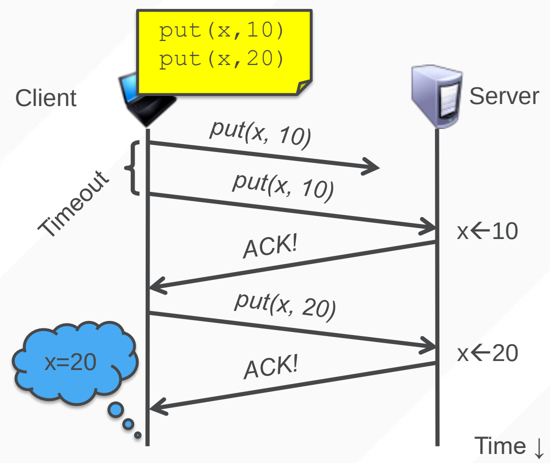
AT-LEAST-ONCE AND SIDE EFFECTS

- Client sends a “debit $10 from bank account” RPC



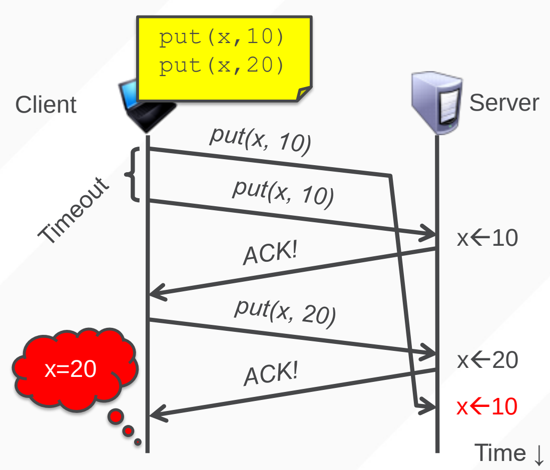
AT-LEAST-ONCE AND WRITES

- put(x, *value*), then get(x): expect answer to be *value*



AT-LEAST-ONCE AND WRITES

- Consider a client storing **key-value pairs** in a **database**
  - put(x, *value*), then get(x): expect answer to be *value*



SO IS AT-LEAST-ONCE *EVER* OKAY?

- **Yes:** If they are read-only operations with no side effects
  - *e.g.*, read a key's value in a database
- **Yes:** If the application has its own functionality to cope with duplication and reordering
  - You will implement this in Project 2

AT-MOST-ONCE SCHEME

- **Idea:** server RPC code detects duplicate requests
  - Returns previous reply **instead of re-running handler**
- *How to detect a duplicate request?*
  - **Test:** Server sees same function, same arguments twice
    - **No!** Sometimes applications **legitimately** submit the same function with same augments, twice in a row

AT-MOST-ONCE SCHEME

- *How to detect a duplicate request?*
  - Client includes unique ***transaction ID*** (***xid***) with each one of its RPC requests
  - Client uses **same xid** for retransmitted requests

```
At-Most-Once Server
if seen[xid]:
    retval = old[xid]
else:
    retval = handler()
    old[xid] = retval
    seen[xid] = true
return retval
```

AT MOST ONCE: ENSURING UNIQUE XIDS

- ***How to ensure that the xid is unique?***

1. Combine a unique client ID (*e.g.*, IP address) with the current time of day
2. Combine unique client ID with a sequence number
   - Suppose the client crashes and restarts. *Can it reuse the same client ID?*
3. Big random number

AT-MOST-ONCE: DISCARDING SERVER STATE

- **Problem:** `seen` and `old` arrays will **grow without bound**
- **Observation:** By construction, when the client gets a response to a particular xid, it will **never re-send it**
- Client could **tell** server "I'm done with xid *x* – delete it"
  - Have to tell the server about **each and every** retired xid
    - Could **piggyback** on subsequent requests

Significant overhead if many RPCs are in flight, in parallel

AT-MOST-ONCE: DISCARDING SERVER STATE

- **Problem:** `seen` and `old` arrays will **grow without bound**
- Suppose xid = ⟨unique client id, sequence no.⟩
  - *e.g.* ⟨42, 1000⟩, ⟨42, 1001⟩, ⟨42, 1002⟩
- Client includes "seen all replies ≤ *X*" with every RPC
  - Much like TCP sequence numbers, acks
- *How does the client **know** that the server received the information about retired RPCs?*
  - Each one of these is cumulative: later seen messages subsume earlier ones

AT-MOST-ONCE: CONCURRENT REQUESTS

- **Problem:** How to handle a duplicate request while the original is still executing?
  - Server doesn't know reply yet. Also, we don't want to run the procedure twice
- **Idea:** Add a **`pending`** flag per executing RPC
  - Server waits for the procedure to finish, or ignores

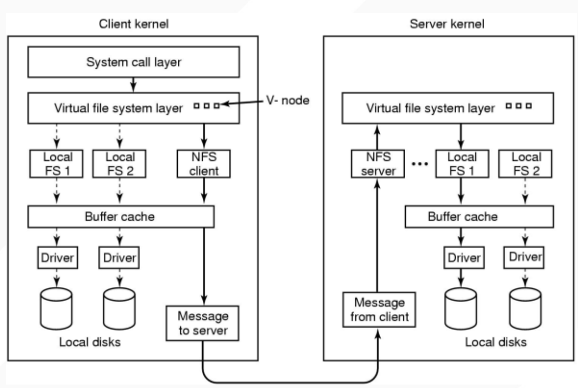
AT MOST ONCE: SERVER CRASH AND RESTART

- **Problem:** Server may crash and restart
- *Does server need to write its tables to disk?*
- Yes! On **server crash and restart:**
  - If **`old[]`**, **`seen[]`** tables are only in memory:
    - Server will forget, **accept duplicate requests**

RPC SEMANTICS

Delivery Guarantees			RPC Call Semantics
Retry Request	**Duplicate Filtering**	**Retransmit Response**	
No	NA	NA	*Maybe*
Yes	No	Re-execute Procedure	*At-least once*
Yes	Yes	Retransmit reply	*At-most once*

SUMMARY: RPC



- RPC everywhere!
- **Necessary** issues surrounding machine heterogeneity
- **Subtle** issues around handling **failures**

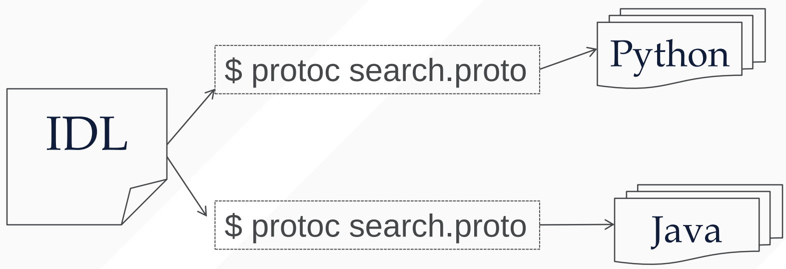
Outline

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) {}
}
```

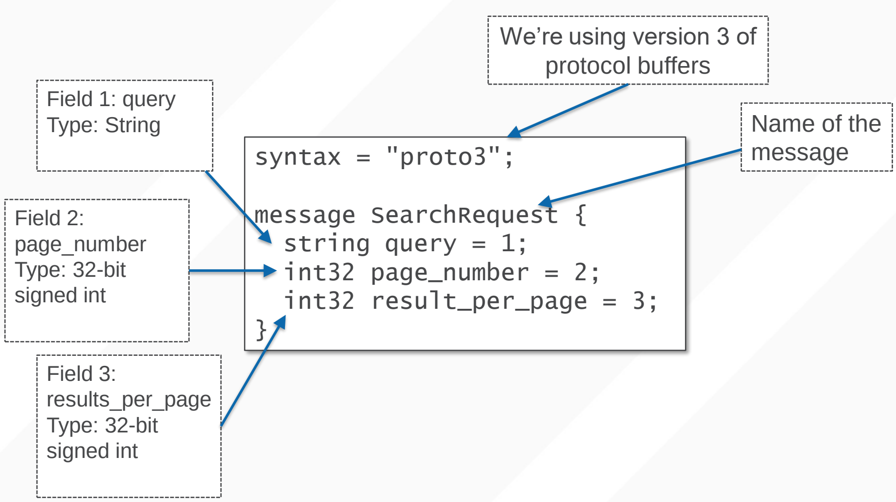
IDL: INTERFACE DEFINITION LANGUAGE



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

IDL LANGUAGE: PROTOCOL BUFFERS

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



PROTOCOL BUFFERS: BASE TYPES

- protobuf IDL:
  - double, float
  - int32, int64
  - uint32, uint64
  - bool
  - string
  - bytes
- Python:
  - float, float
  - int, int/long
  - int, int/long
  - bool
  - str
  - str
- Java:
  - double, float
  - int, long
  - int, long
  - Boolean
  - String
  - ByteString
- C++:
  - double, float
  - int32, int64
  - uint32, uint64
  - bool
  - string
  - 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;
}
```

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

JAVA OVERVIEW: THE SERVER

- Outer class (e.g., ATMServer)
  - start(): starts the server (listens for incoming connections)
  - stop(): stops the server
  - blockUntilShutdown(): internal method
  - main(): reads command-line arguments, calls start()
  - static class ATMServerImpl extends ATMServerGrpc…
    - This is where you specify the implementation of the RPCs

ATMSERVERIMPL

```
message SearchRequest {
        string query = 1;
}

message SearchResponse {
        int32 numResponses = 1;
        repeated string response = 2;
}

service Search {
  rpc searchWeb(SearchRequest) returns (SearchResult) {}
}
```

- Implement each RPC here by overriding the compiler-generated code
- Compiler output goes in target/generated-sources folder
  - Generated-sources/java contains ‘message’ definitions
  - Generated-sources/grpc-java contains ‘service’ definitions

HOW TO GET STARTED?

1. Define your messages and services in proto file
2. Run the compiler

   $ mvn protobuf:compile protobuf:compile-custom

3. Find proc signature in …/ATMServerGrpc.java

```
  /**
   */
  public static abstract class ATMServerImplBase implements io.grpc.BindableServ
ice {

    /**
     */
    public void ping(atm.PingRequest request,
        io.grpc.stub.StreamObserver<atm.PingResponse> responseObserver) {
      asyncUnimplementedUnaryCall(METHOD_PING, responseObserver);
    }

    /**
     */
    public void searchWeb(atm.SearchRequest request,
        io.grpc.stub.StreamObserver<atm.SearchResponse> responseObserver) {
      asyncUnimplementedUnaryCall(METHOD_SEARCH_WEB, responseObserver);
    }
```

4. Override in src/main/java/atm/ATMServer.java

UNDERSTANDING STUB CODE

```
service Search {
   rpc searchWeb(SearchRequest) returns
(SearchResult) {}
}
```

- Yet stub code looks like this:

```
  /**
   */
  public static abstract class ATMServerImplBase implements io.grpc.BindableServ
ice {

    /**
     */
    public void ping(atm.PingRequest request,
        io.grpc.stub.StreamObserver<atm.PingResponse> responseObserver) {
      asyncUnimplementedUnaryCall(METHOD_PING, responseObserver);
    }

    /**
     */
    public void searchWeb(atm.SearchRequest request,
        io.grpc.stub.StreamObserver<atm.SearchResponse> responseObserver) {
      asyncUnimplementedUnaryCall(METHOD_SEARCH_WEB, responseObserver);
    }
```

- Void return type
- Always two arguments
  - 1st argument is a single message which is the only argument the RPC can take
    - SearchRequest in this case
  - 2nd argument is an “in-out” argument used to return data back to the caller
- gRPC defines four types of in-out return types—we’re only using the most basic one
- To return data to client, you pass it to the “responseObserver”

CONSTRUCTING RESULTS FROM STUB CODE

1. Construct an object based on the return type

- `Example:`
  - `SearchResultBuilder srb;`
  - `srb.setQuery("foo bar");`
  - `SearchResult sr = srb.build();`

RETURNING RESULTS FROM STUB CODE

1. Construct an object based on the return type
2. Pass to the responseObserver
3. Tell the responseObserver you're done

- `Example:`
  - `responseObserver.onNext(sr);`
  - `responseObserver.onCompleted();`

ERRORS AND EXCEPTIONS

- The server can throw an Exception, which is translated into an Exception in the client
  - Catch try…catch and handle as appropriate

DIVING DEEPER

- https://grpc.io/
- https://grpc.io/docs/quickstart/java.html
- https://grpc.io/docs/tutorials/basic/java.html
- https://grpc.io/docs/reference/java/generated-code.html

Outline

1. RPC fundamentals
2. Handling failures in RPCs
3. gRPC: Google RPC overview
4. ATM Server demo

SIMPLE ATM SERVER

- Operations:
  - login
    - Account number + PIN
  - deposit
    - $$$
  - getBalance
  - logout

SIMPLE ATM SERVER

- Keeping track of account + pin with "login tokens"
- After logging in, get a token
- Use token to deposit money, withdraw, transfer, …

UC San Diego