Making Remote Calls

Remote procedure calls and their infrastructure

Overview

- Call Versions (local, inter-process, remote)
- Mechanics of Remote Calls
  - Marshaling/Serialization
  - Data representation
  - Message structure and Schema Evolution
  - Interface Definition Language
  - Tooling: generators
- Cross language call infrastructures (Thrift, gRPC)
- Next: Distributed Objects (CORBA, RMI)

Exercise: Make a Remote Call!

```
#include “foo.h”
Int i=5;
Char * c=”Hello World”;
Main (argc, argv) {
   Int r = foo(i,c);
}
```

File caller.c on Host A

```
#include “foo.h”
Int foo (int x, char* y){
   Return (strlen(y) > x) ? 0 : 1;
}
```

File service.c on Host B

Create software that executes main on A and uses function foo on B! All you have is the socket API.

Call Versions

- local calls
- Inter-process calls
- Remote calls

Remote Calls vs. Remote Messages

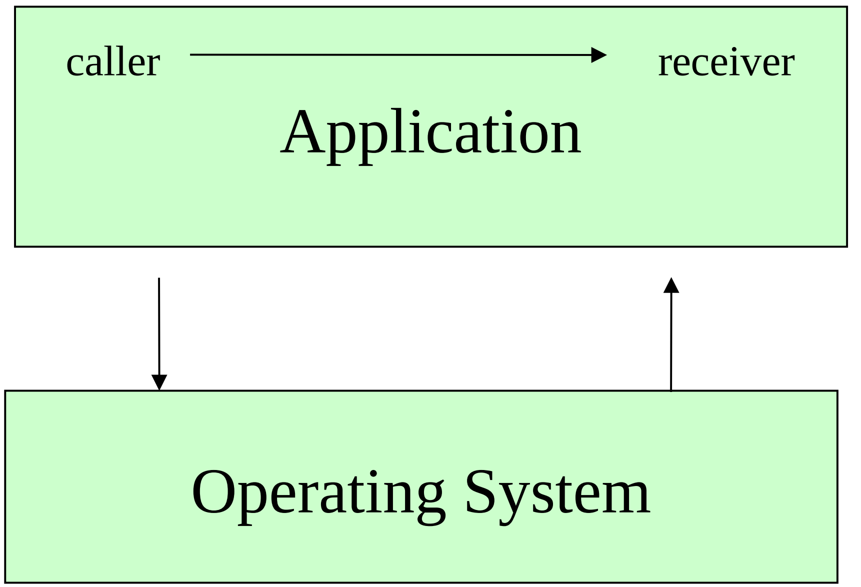
Ret = foo (int I, char * s)

Socket.send(char * buffer)

Call based middleware hides remote service calls behind a programming language call. Tight coupling and synchronous processing are often a consequence of this approach!

Message based middleware creates a new concept: the message and its delivery semantics. A message system can always simulate a call based system but not vice versa.

Local, In-Process Calls



As long as we stay within one programming language no special middleware is required. Calls into the OS are not Inter-process calls. But: Cross-language calls within one process need special attention (e.g. calls to native code in Java)

Local Calls

stack

Caller pushes return address and parameters on stack

1122

fff0 →

Dff0 (return addr)

receivers's stack

data

helloworld

Address:0xFFF0

charptr = 0xFFF0;

intvalue= 0x1122

Callee de-references character pointer. Result is stored in some register. After processing goes back to caller through return address

code

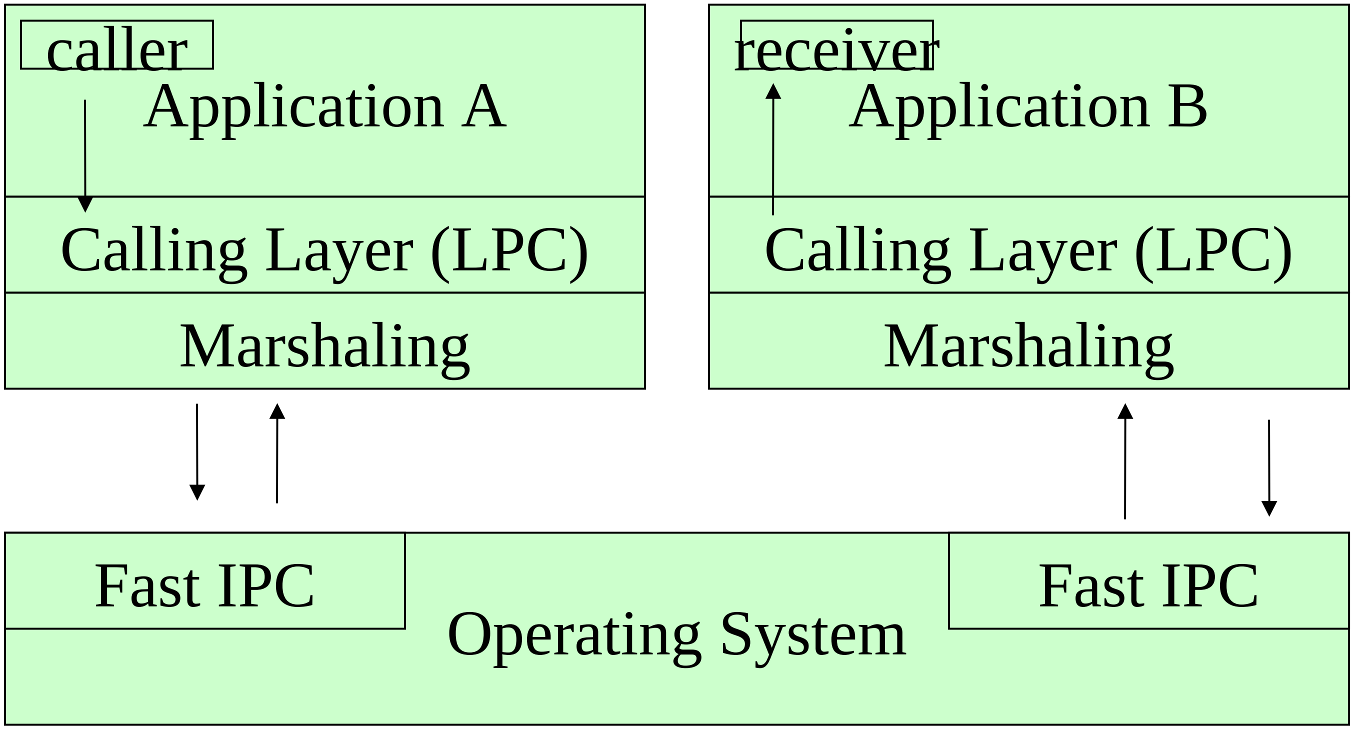
```
Char * charpointer = "SOMESTRING";

Int intvalue = 0x1122;

Main () {

Int result = function(charpointer, intvalue);

Return:Address:0xDFF0
```

```
int result function(charpointer, intvalue) {

Print(charpointer);

Intvalue++;

Return 0;          // store 0 in register X

                   // make a "return"
```

In-Process calls

- Fast (how fast actually?)
- Performed with exactly once semantics
- Type and link safe (but dll and dynamic loading problems)
- Either sequential or concurrent (we decide it!)
- Can assume one name and address space
- Independent of byte ordering
- Controlled in their memory use (e.g. garbage collection)
- Can use value or reference parameters (reference = memory address)
- Transparent programming language “calls” and not obvious messages

Local Interprocess Communication



Find application and function

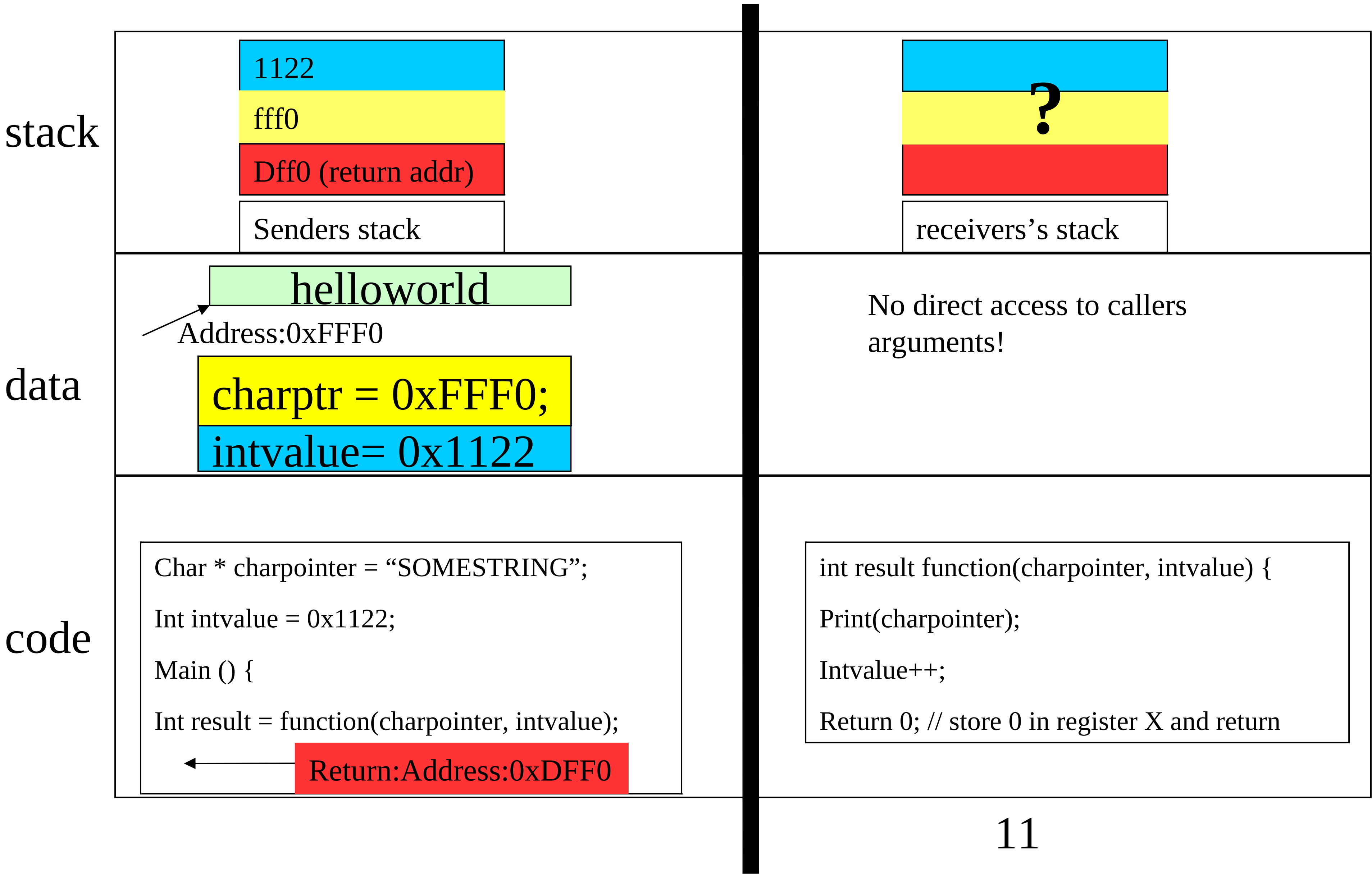
Flatten reference parameters

Some systems use a highly optimized version of RPC called IPC for local inter-process communication. See e.g. Helen Custer, inside Windows NT, chapter “Message passing with the LPC Facility”

Local Inter-process calls

- Pretty fast
- No more exactly once semantics
- Type and link safe if both use same static libraries (but dll and dynamic loading problems)
- Sequential or concurrent (caller does no longer control it! Receiver needs to protect himself)
- Can no longer assume one name and address space
- Still Independent of byte ordering
- Would need cross-process garbage collection
- Can only use value parameters (target process cannot access memory in calling process)
- No longer real programming language “calls”. The missing features must be created through messages

Interprocess Calls

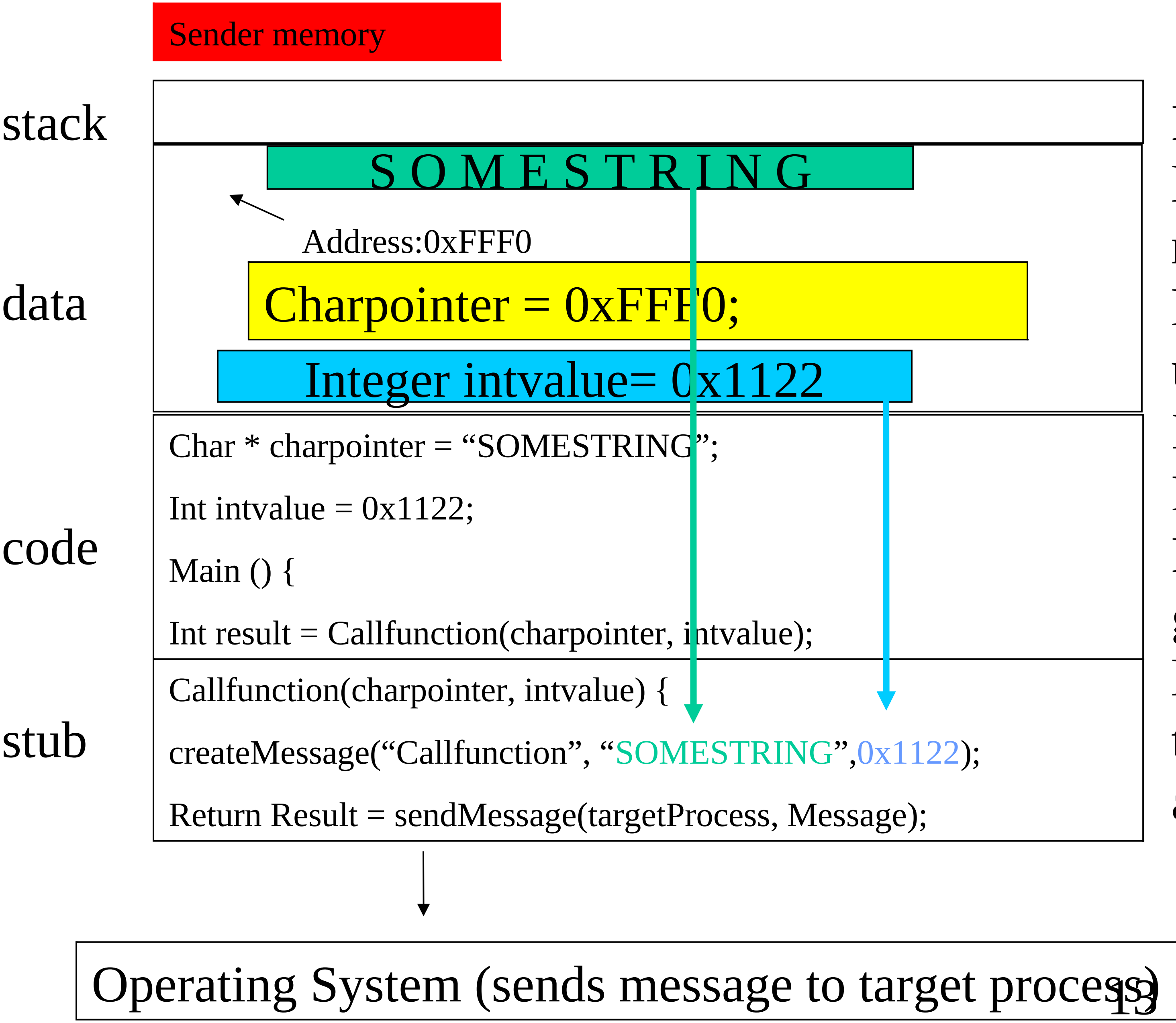


Inter-Process is not local!

- Latency
- Memory Barriers
- Process failures

The good news: same hardware and language at sender and receiver, fewer security problems, a system crash affects both sender and receiver (fail-stop semantics)

Local Inter-process call: Sender



Marshalling layer flattens references.Usually automated using an Interface Definition Language plus generator. LPC layer selects target process and function.

Local Inter-process call: Receiver

receiver memory

Stack

S O M E S T R I N G

Address:0xAFF0

Charpointer = 0xAFF0;

Integer intvalue= 0x1122

data

code

Char * charpointer = “SOMESTRING”;

Int intvalue = 0x1122;

Main () {

Int result = Callfunction(charpointer, intvalue);

skeleton

CallfunctionSkeleton(message) {

Char * charpointer = getArg1(message); intvalue = getArg2(message);

Return Callfunction(charpointer, intvalue);

Which return address is on the stack?

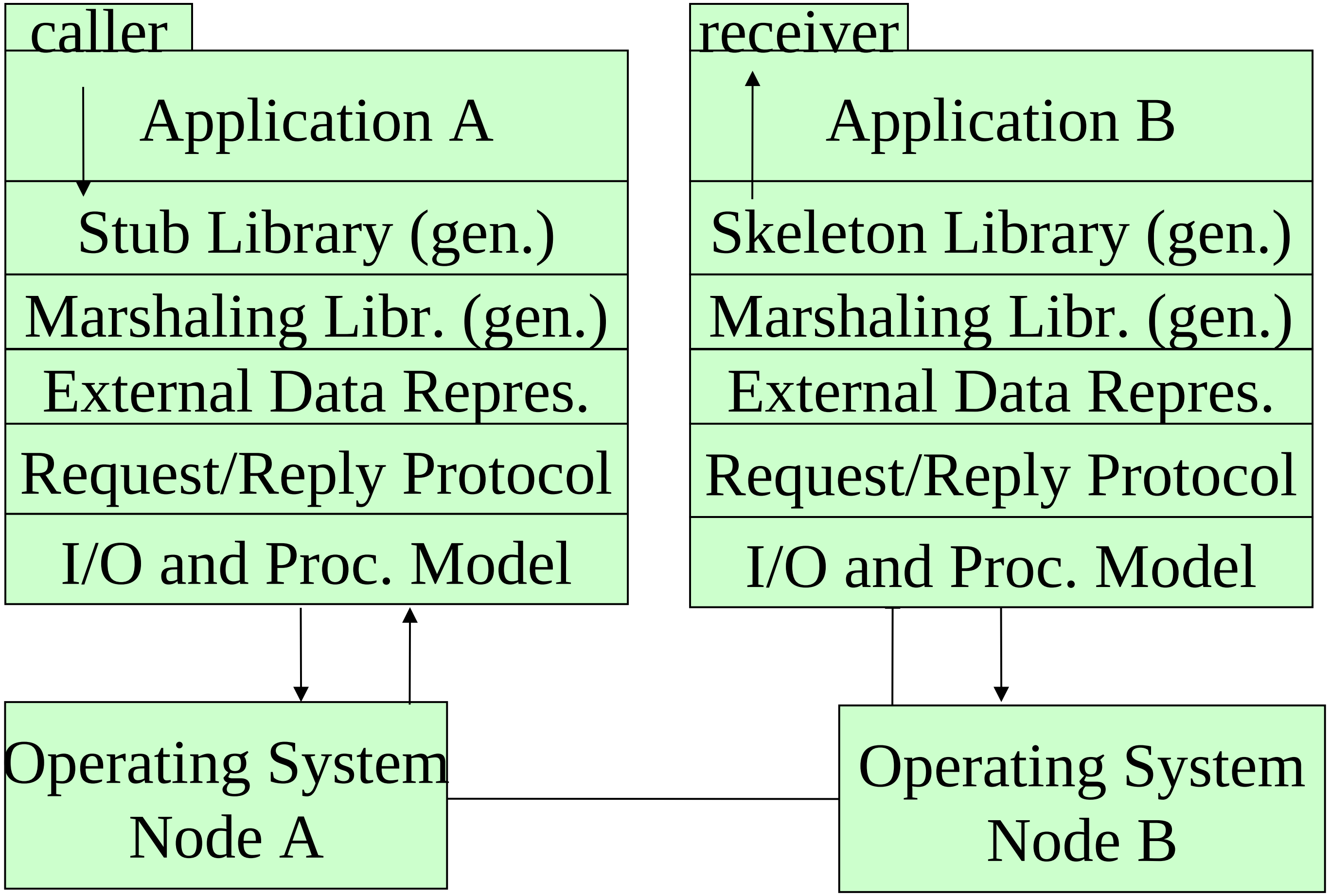
Marshalling layer unpacks message and calls real function.

Operating System (sends message to target process). Returns result to calling process

Remote calls are:

- Much slower than both local versions
- No delivery guarantees without protocol
- Version mismatches will show up at runtime
- Concurrent (caller does no longer control it! Callee needs to protect himself)
- Can no longer assume one name and address space
- Affected by byte ordering
- In need of network garbage collection (if stateful)
- Sometimes Cross-language calls
- Can only use value parameters (target process cannot access memory in calling process)
- No longer programming language “calls”. The missing features must be created through messages
- Frequently stateless

Remote Procedure Calls



Proxy behavior, prog. Lang. Call to message

Serialization

Endian-ness, format

delivery guarantees, e.g. at most once!

The main components of a RPC system. Not shown is the processing framework (threading, async. Etc.). Stub/skeleton libraries are generated from interface definitions.

Mechanics of Remote Calls

- Marshaling/Serialization: maps program data to output format (binary or text)
- External Data-Representation: canonical output format for binary data
- Interface Definition: Defines a Service
- Message Structure and Evolution
- Compilers: generate Stub/Skeleton or Proxy
- Request/Reply protocol: deals with errors
- Process/I/O layer: handles threads and I/O

Marshaling/Serialization

Definition: flattening parameters (basic types or objects) into a common transfer format (message). The target site will do the transformation from the transfer format into the original types or objects

- Language dependent output format (priopprietary, sometimes slow, limits in expressiveness
- Language independent output format (sometimes bloated, verbose)
- Binary Schema based (sender and receiver know structure of every message, I.e. which type/variable is at what offset, function names replaced with numbers, variable data length encoding, compression)
- Binary self describing (the transfer format contains type and variable information as well. Needs some flexible capabilities of the involved languages
- Textual, self describing (XML representation of types or objects, e.g. using SOAP)
- Textual with schema for reader/writer. Allows advanced schema evolution and dynamic serializations

The typical trade-off between speed (binary) and flexibility (self-describing) which allows e.g. to skip unknown parts.

Serialization to Text

Class Person {
String user_name = new string(“Martin”);
Int favourite_number = 1337;
String [] interests = new array [“daydreaming”, “hacking”;
}

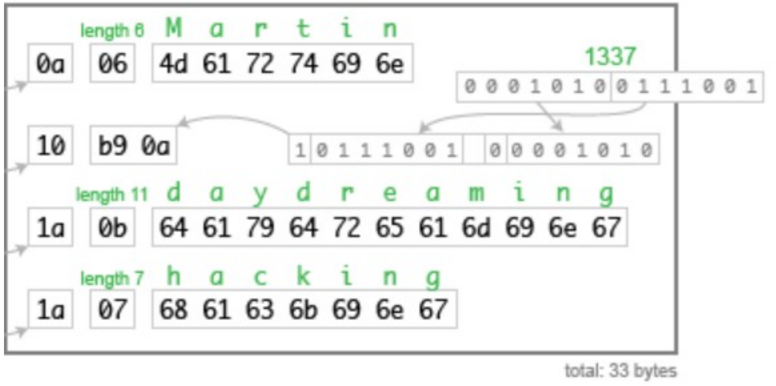
```
{
    "userName": "Martin",
    "favouriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

Less compact than binary. Watch out for language limits (int/floating point) in Javascript. XML allows language independent encoding.
After:https://martin.kleppmann.com/2012/12/protobuf.png

Serialization to Binary

Class Person {
String user_name = new string("Martin");
Int favourite_number = 1337;
String [] interests = new array ["daydreaming", "hacking";
}

010064d6172749663000...



Compact but requires schema allows language independent encoding.
After:https://martin.kleppmann.com/2012/12/protobuf.png

Example (Generated) Code

- Marshalling: Disassemble data structures into transmittable form

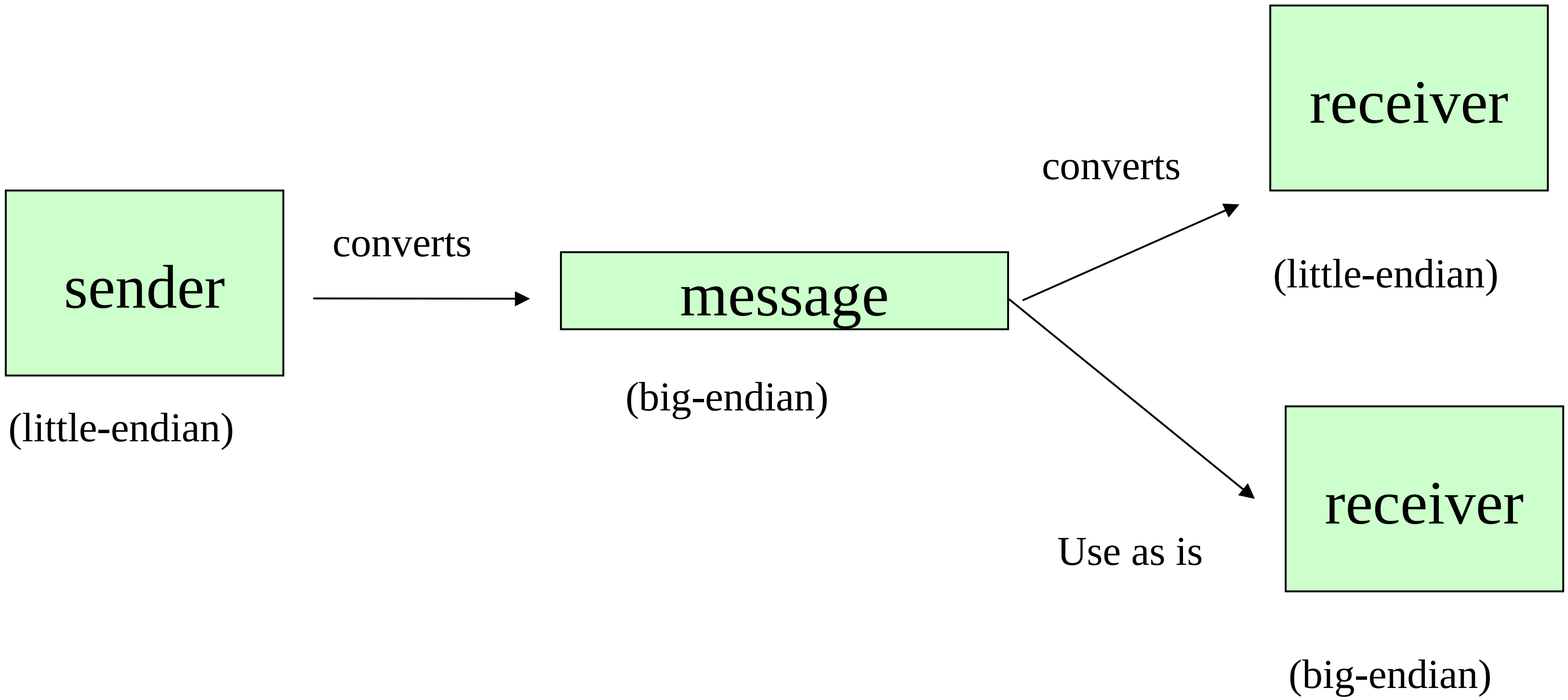
```
char * marshal() {
 char * msg;
 msg=new char[4*(sizeof(int)+1) +
              strlen(name)+1];
 sprintf(msg,"%d %d %d %d %s",
         dob.day,dob.month,dob.year,
         strlen(name),name);
 return(msg);
};
```

- Unmarshalling: Reassemble the complex data structure.

```
void unmarshal(char * msg) {
 int name_len;
 sscanf(msg,"%d %d %d %d ",
        &dob.day,&dob.month,
        &dob.year,&name_len);
 name = new char[name_len+1];
 sscanf(msg,"%d %d %d %d %s",
        &dob.day,&dob.month,
        &dob.year,&name_len,name);
};
```

From: W.Emmerich

External Data Representation



Using a standard network byte-order (big-endian here) results in some unnecessary conversions between little-endian hosts. What is the big advantage compared with a "use sender format" policy? (Hint: think about new systems)

Request-Reply Message Structure

Message Type
(request or reply)

Request ID
e.g. 5 – the fifth request

Needed for request-reply layer and delivery guarantees

Object Reference of remote object
(if RMI)

Method ID/Procedure ID
(what function/method to call)

Used by the remote dispatcher to create call to proper method or function

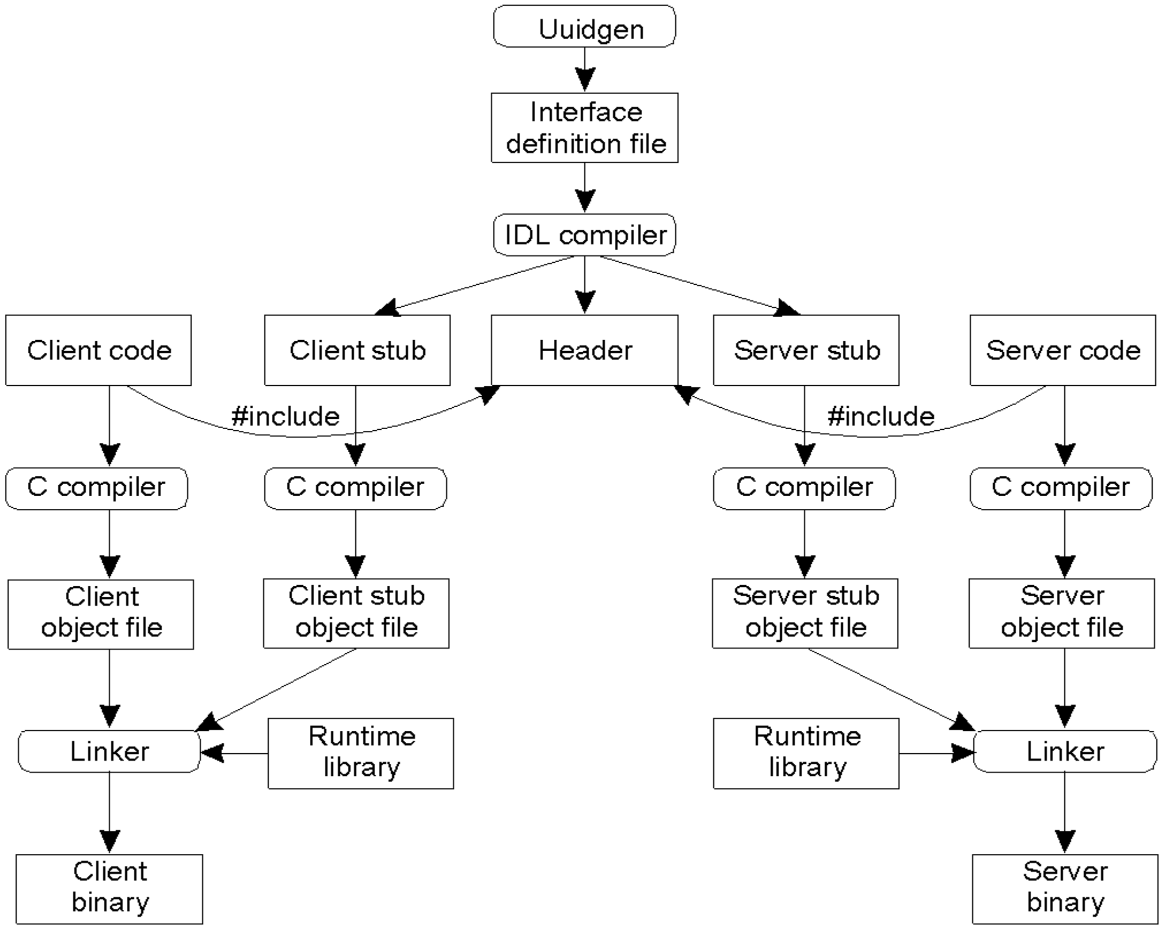
Optional: fields for authentication e.g. client credentials

Interface Definition (Unix RPCs)

```
const NL=64;
struct Player {
 struct DoB {int day; int month; int year;}
 string name<NL>;
};
program PLAYERPROG {
 version PLAYERVERSION {
  void PRINT(Player)=0;
  int STORE(Player)=1;
  Player LOAD(int)=2;
 }= 0;
} = 105040;
```

From W.Emmerich, Engineering Distributed Objects; Compare with Webservices WSDL format, REST, Thrift, gRPC, XML-RPC etc.!

Generated: Stub/Skeleton



The steps in writing a client and a server in DCE RPC. (from van Steen, Tanenbaum, Distributed Systems)

What if Data or Functions change?

- with many clients in the field, different versions need to coexist

- forward compatibility is required: older receivers need to understand messages from newer senders

- backward compatibility is required: newer receivers need to unterstand messages from older senders

Wherever different senders or receivers cooperate, schema evolution becomes an issue (databases, message queues, RPC)

Schema Evolution

Interface Definition:

Struct X { 1:optional int Y, default: 0

2:required string Z

3:optional smallint W}

Function A {1:optional “put”, void, string}

Function B { 2: required “get”, string, void}

Many serialization libraries allow the tagging of data or functions with “optional” or “required”. They also require unique numbers for data and functions within definitions. Some like AVRO provide complete schemas for reader and writer and allow dynamic matching . See https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

Exercise: What breaks compatibility?

	Forward compatible	Backward compatible
Change opt.->req.		
Change req.->opt		
Add new req.data		
Add new opt data		
Change funct. #		
Add opt. function		
Add req. function		
Add data with default		
Change data size		
Change funct.order		
Change data order		
Remove data type in encoding		

Stubs and Skeletons

Generated in advance from IDL file

Generated on demand from class file

Distributed in advance to all clients/servers

Downloaded on demand

There are endless ways to generate stubs and skeletons. Statically or dynamically with the help of generators.

Delivery guarantees revisited

Local /remote	Retransmit	Filter Duplicates	Request	Semantics
Remote	N	N/A	N/A	maybe/ Best effort
Remote	Y	N	Re-execute request	At least once
Remote	Y	Y	Re-transmit reply	At most once
Local - no persistence	N/A	N/A	N/A	Exactly once

Idempotent operations

Definition:

If you can send a request a second time without breaking application semantics if the request was already executed the first time it was sent – then this operation is idempotent.

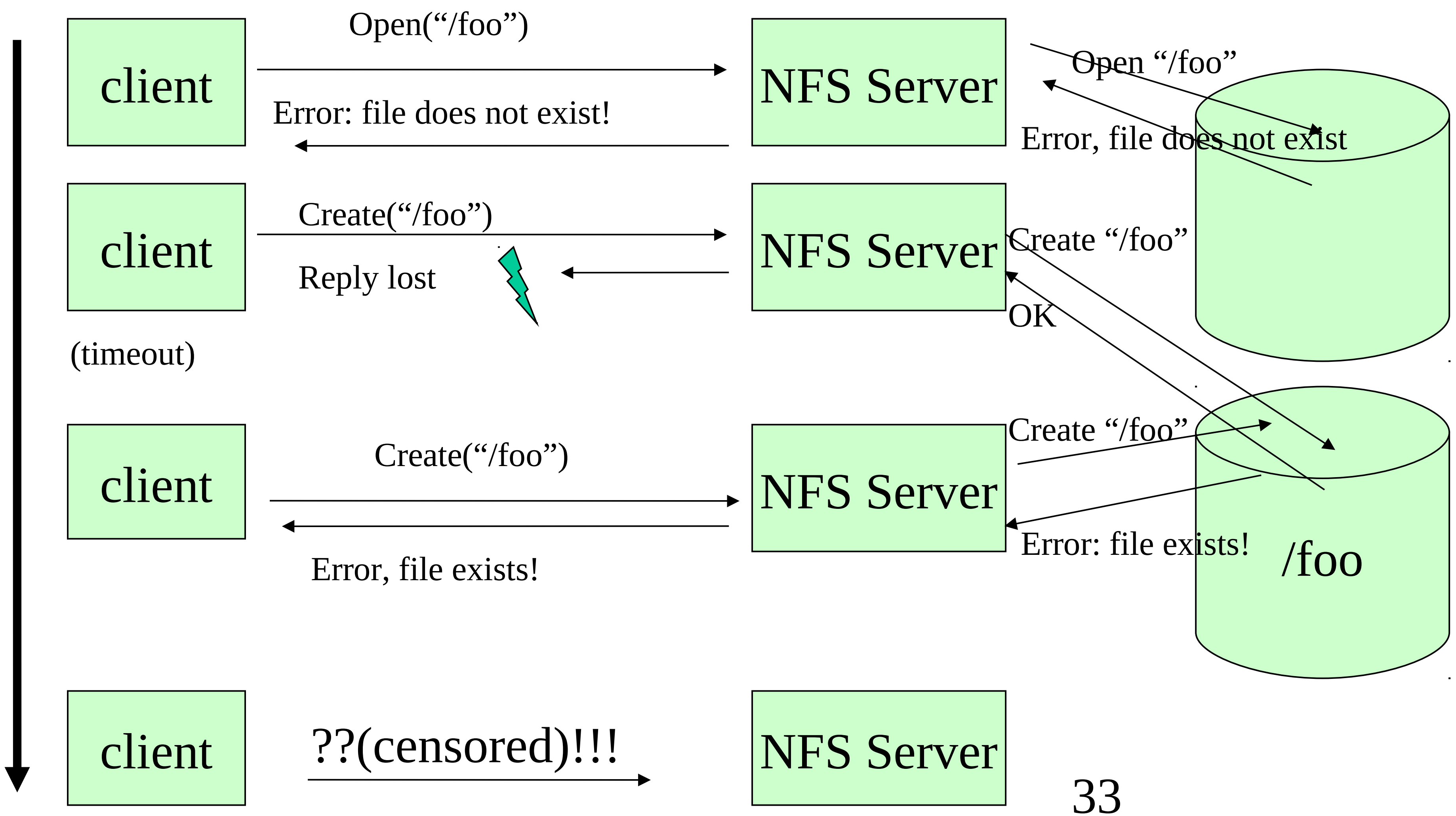
Example: http “get” request. (page counter does NOT break application semantic)

With idempotent operations you can build a request/reply protocol using only at-least-once semantics!

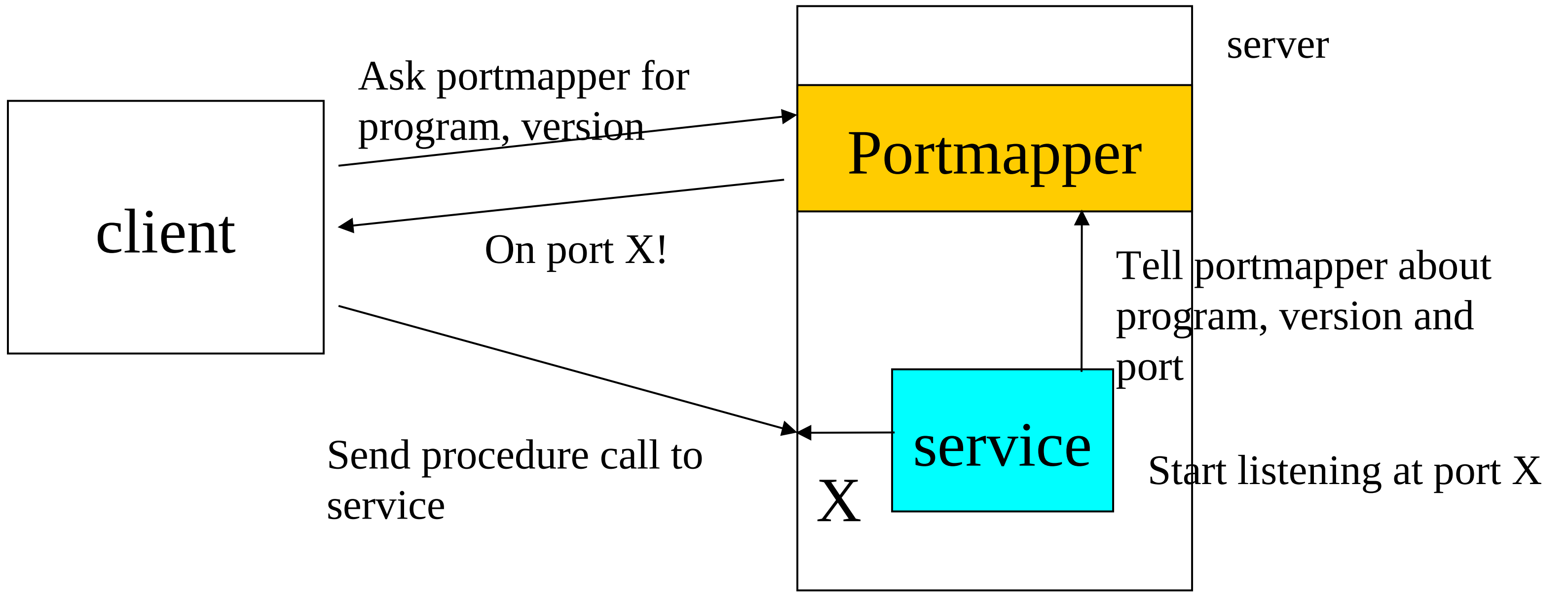
If operation is NOT idempotent:

- Use message ID to filter for duplicate sends
- Keep result of request execution in a history list on the server for re-transmit if reply was lost.
- Keeping state on the server introduces the problem of how long to store old replies and when to scrap them.
- Frequently used: client “leases” for server side resources

SUN-NFS: at least once semantics without idempotent operations



Finding a RPC server



This is called "binding" and can be handled in different ways (inetd, DCE, Unix portmapper)

Cross-Language Call Infrastructure

- CORBA
- Microsoft CLR
- Thrift
- Google Protocol Buffers and gRPC

Remote Cross Language Messages

IDL file:
Structure Foo {
  Var1 x; Var2 y;
  Enum z {…}
}

RPC-Compiler

.java

Foo.get_x()
Foo.get_(y)
Serialization
Reflection

Runtime Framework (encodings, Transports, tracing)

.cpp

Foo->get_x()
Foo->get_(y)
Serialization
Reflection

Important Questions

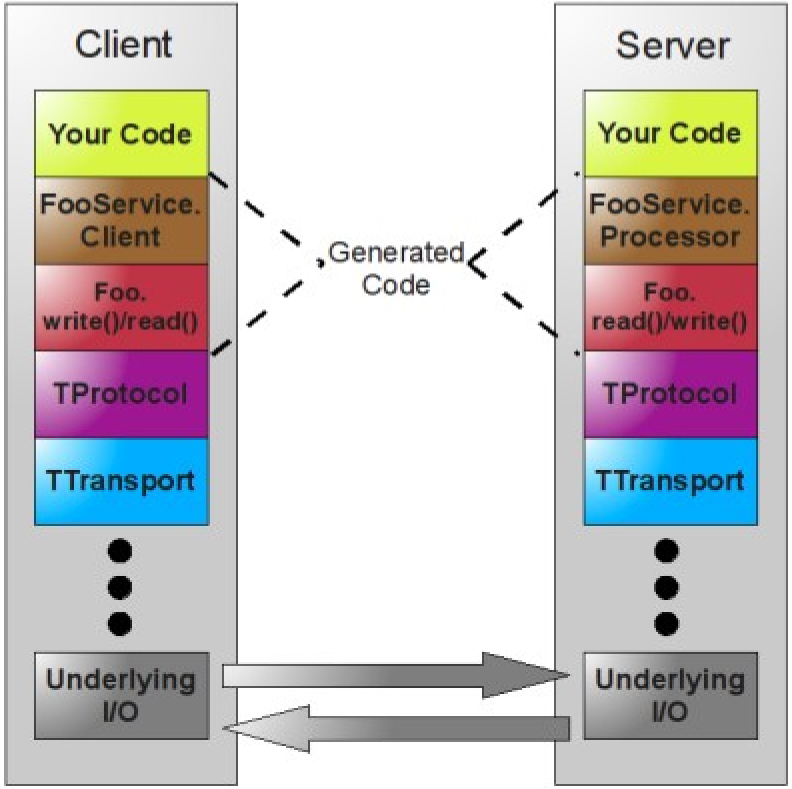
- Are data types easily expressed using the IDL?
- Is hard or soft versioning used?
- Are structures self-describing?
- Is it possible to change the structures later and keep backward compatibility?
- Is it possible to change processing of structures later and keep forward compatibility?
- Are there bindings for all languages in use at my company?
- Do I need different encodings (binary/textual)?
- Does changing serialization require a re-compile?
- Can I extend/change the runtime system (e.g. add trace statements)?

Apache Thrift

- Simple Interface Definition Language
- Efficient Serialization in Space and Time
- Variable Protocols
- Support for different Languages
- Code Generators for Glue Code
- Soft Versioning to allow interface and data type evolution between teams

Designed by Facebook, now an Apache project.

Thrift Protocol Stack



From:; A.Prunicki, Thrift Overview, http://jnb.ociweb.com/jnb/jnbJun2009.html

Google Protocol Buffers

.proto file:

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

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default
= HOME];
  }  repeated PhoneNumber phone = 4;
}
```

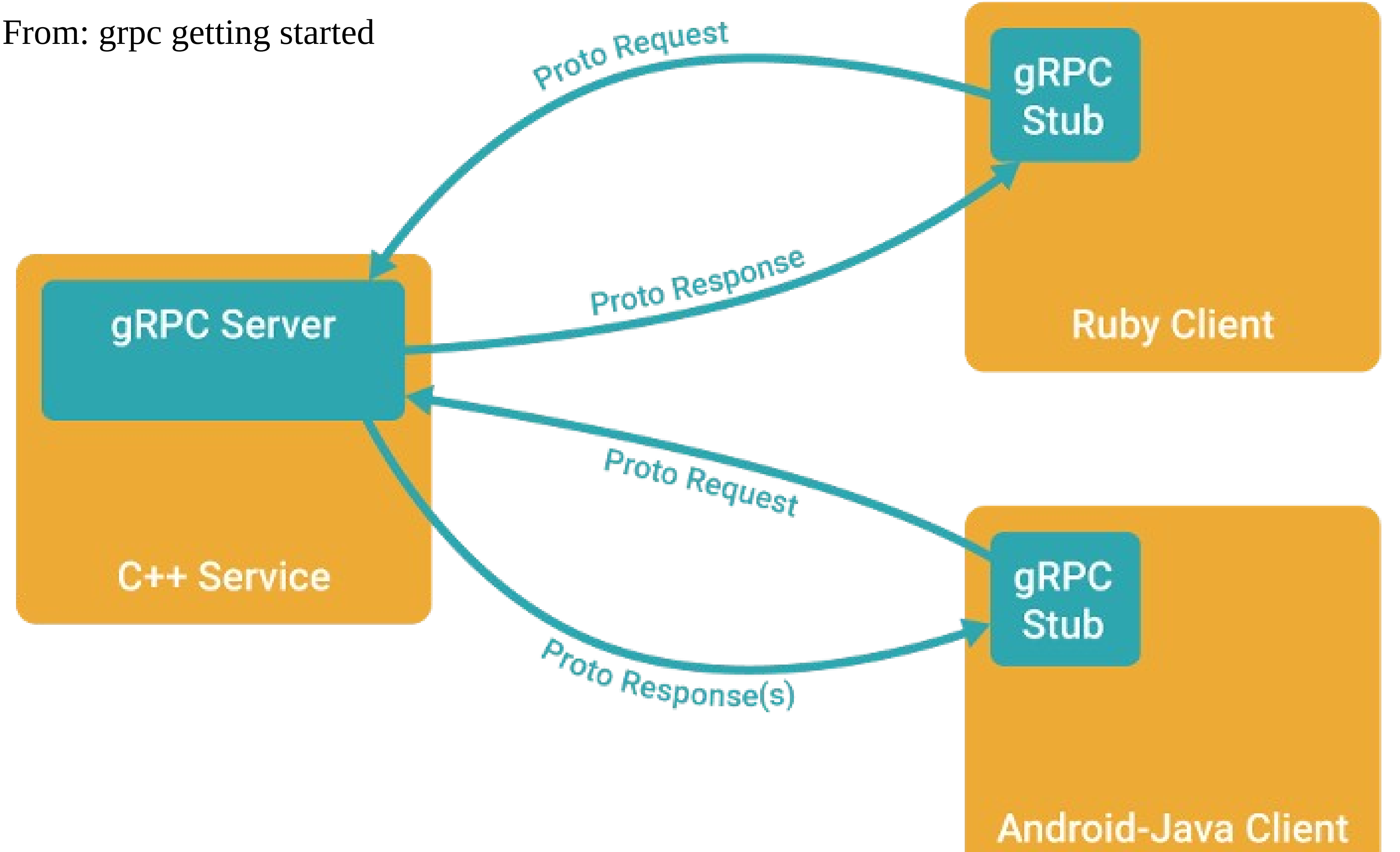
.cpp file:

```
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);
```

From: protocol buffers developers guide:
http://code.google.com/apis/protocolbuffers/docs/overview.html

GRPC

From: grpc getting started



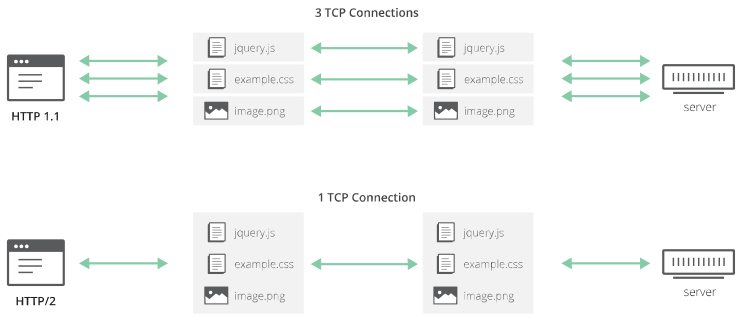
gRPC-Web

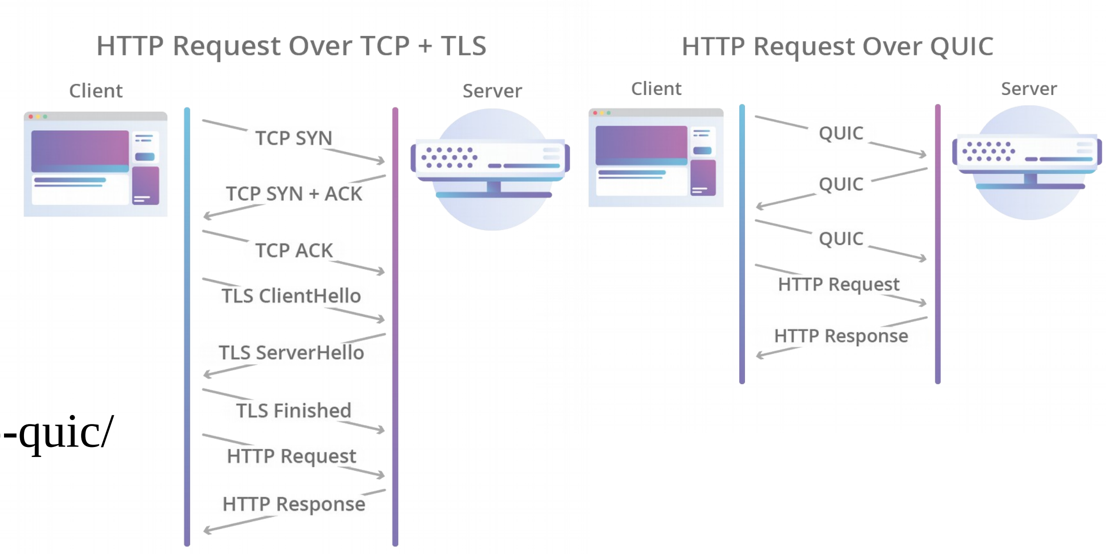
gRPC-Web enables you to define the service "contract" between client web applications and backend gRPC servers using **.proto** definitions and auto-generate client JavaScript (you can choose between Closure compiler JavaScript or the more widely used CommonJS). What you get to leave out of the development process: creating custom JSON serialization and deserialization logic, wrangling HTTP status codes (which can vary across REST APIs), content type negotiation, etc.

From a broader architectural perspective, what gRPC-Web makes possible is end-to-end gRPC. The diagram below illustrates this

https://www.cncf.io/blog/2018/10/24/grpc-web-is-going-ga/

The Future: quic/http3





https://blog.cloudflare.com/the-road-to-quic/

iAPX / Smack-Fu Master, in training POPULAR NOV 13, 2018 12:25 AM

The problem is not HTTP over TCP, it is out-sourced Ads and trackers.

67 requests, 1.0MB transferred, 1.48s with AdBlock Plus and my own Tracker Blocking Chrome Extension.
263 requests, 2.8MB transferred, 26.92s without any of that.
This is the home page of Ars Technica as I write it (full reload).

Do you think that HTTP/3 will generate 4X less request, will be 3X faster to transfer and will be 18X faster to complete loading?!?

This is not an HTTP problem, this a website design and abuse of ads and tracker problem 😉
Google is trying to solve or hide a Google created problem.

Last edited by iAPX on Mon Nov 12, 2018 7:18 pm

+249 (+263 / -14) 22 posts | registered 3/18/2018

https://arstechnica.com/gadgets/2018/11/the-next-version-of-http-wont-be-using-tcp/?comments=1&post=36350073

A Critique of RPCs

- Should RPCs really look like normal calls? (Im Waldo, A note on distributed computing)
- Difficulty in recovery after malfunction or error. For instance, do we rollback or throw exceptions? How do we handle these errors? Can we just try again?
- Difficulty in sequencing operations. If all calls are synchronous and some of these calls can fail, it can require a significant amount of code to ensure correct re-execution to preserve order moving forward.
- Remote Procedure Call forces synchronous programming: a method is invoked and the invoking process waits for a response.
- Backpressure, or blocking on previous actions completing, load-shedding, or dropping messages on the floor when the system is overloaded, and priority servicing become more difficult with the call-and-response model of Remote Procedure Call.
- “There is, in fact, no protocol that guarantees that both sides definitely and unambiguously know that the RPC is over in the face of a lossy network.” Tanenbaum and Renesse (1987)

C. Meiklejohn, Remote Procedure Calls,
https://christophermeiklejohn.com/pl/2016/04/12/rpc.html

Homework

1) Look at Robert Kubis slides on http2, protocol buffers and GRPC http://de.slideshare.net/AboutYouGmbH/robert-kubis-grpc-boilerplate-to-highperformance-scalable-apis-codetalks-2015

2) download GRPC Java examples from http://www.grpc.io/docs/
Read the getting started guide and start compiling the examples.

3) Run server and client and test the runtime.

4) Define your own interface and generate the server and client side

Resources

- John Bloomer, Power Programming with RPC
- John R.Corbin, The Art of Distributed Applications. Programming Techniques for Remote Procedure Calls
- Ward Rosenberry, Jim Teague, Distributing Applications across DCE and Windows NT
- Mark Slee, Aditya Agarwal and Marc Kwiatkowski, Thrift: Scalable Cross-Language Services Implementation
- Thomas Bayer, Protocol Buffers, Etch, Hadoop und Thrift im Vergleich
- Andrew Prunicki, Apache Thrift
- Google Protocol Buffers, https://developers.google.com/protocol-buffers/docs/tutorials
- GRPC getting started: http://www.grpc.io/docs/
- GRPC Java examples: https://github.com/grpc/grpc-java/tree/master/examples
- M. Kleppmann, Designing Data-Intensive Applications, Oreilly Pub.
- M.Kleppmann, https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
- Tyler Treat, Thrift on Steroids: A Tale of Scale and Abstraction, http://bravenewgeek.com/thrift-on-steroids-a-tale-of-scale-and-abstraction/