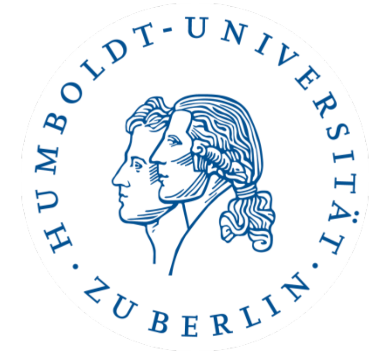


Semesterprojekt
Implementierung eines Brettspiels
(inklusive computergesteuerter Spieler)

Wintersemester 16/17

Remote Procedure Calls

Patrick Schäfer
patrick.schaefer@hu-berlin.de

Marc Bux
buxmarcn@informatik.hu-berlin.de

Public API

- Benefits of a **public** API:
  - platform and language independence (C#, Java, …).
  - support for client/server architecture (multiplayer mode).
  - hide your implementation details (AI).
  - run code on separate servers.

- A **public & common** API provides:
  - reusability of codes (AI usable with all GUIs).

- Options:
  - RPCs: gRPC, Apache Thrift, ...
  - REpresentational State Transfer (REST) with JSON, XML
  - manually writing/reading streams to/from sockets
  - message passing interface (MPI)

Use Case: Address Book

- Suppose we have an address book, which implements the methods:

```
class AddressBook {
	Map<String, Person> addressBook;

	public Person findPerson(String name) {
	  return addressBook.get(name);
	}

	public List<PhoneNumber> listPhoneNumbers(String name) {
	  return addressBook.get(name).getPhoneList();
	} }
```

- We want to access these methods from a remote client.

Remote Procedure Calls

- “A remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in another address space (commonly on another computer on a shared network), […] without the programmer explicitly coding the details for the remote interaction. “ *Wikipedia*

- Typically used for:
  - Client/server architecture: Developing clients which are communicating to a server.
  - Designing a protocol that needs to be accurate, efficient and platform/language independent.
  - Low latency, highly scalable, distributed systems.

RPC

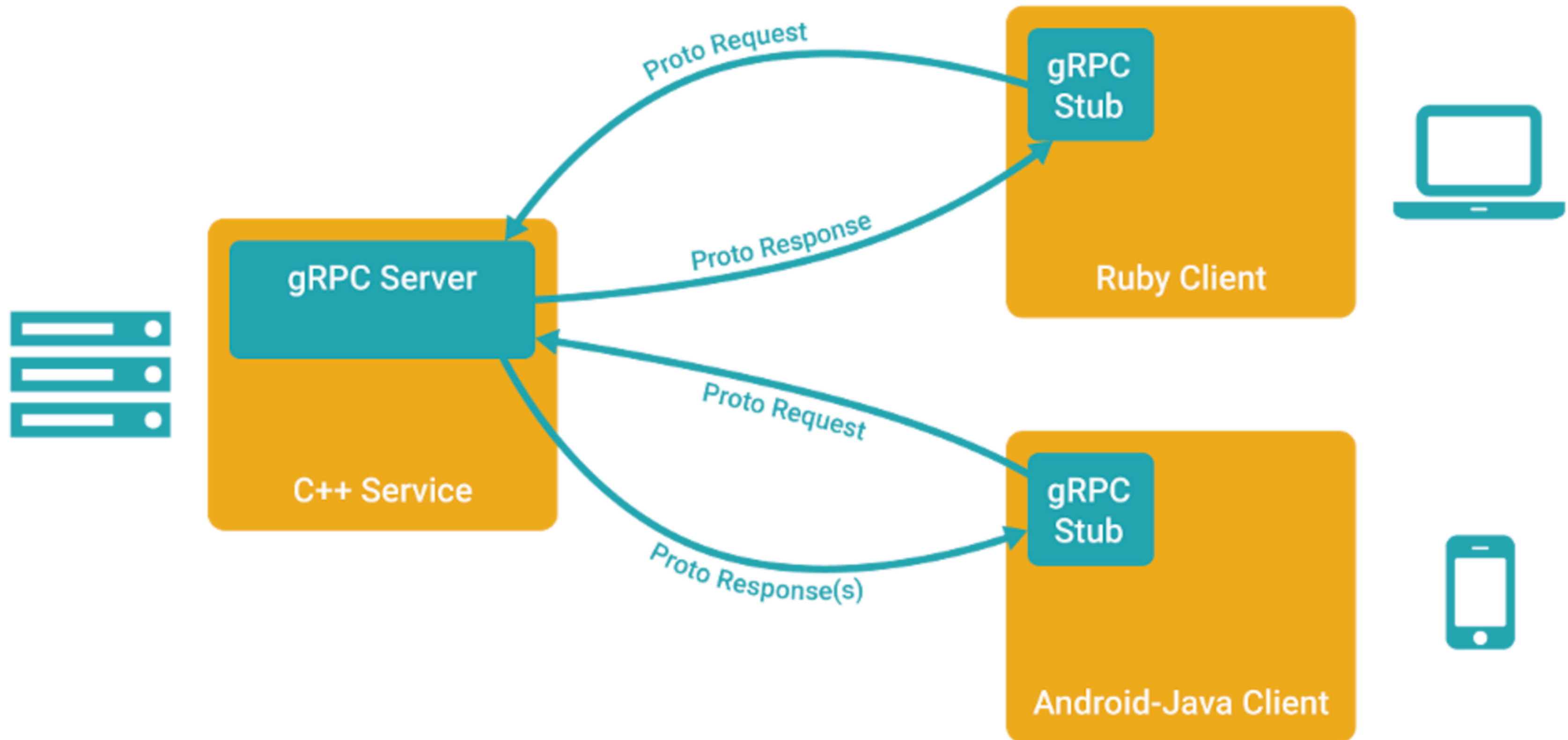
- Local: a call passes the name and the method returns the person’s details.
- RPC: a *client* sends a *request* with the name, then the *server* sends a *response* with the person’s details.

```
class AddressBook {
    Map<String, Person> addressBook;

    public Person findPerson(String name) {
     return addressBook.get(name);
    }

    public List<PhoneNumber> listPhoneNumbers(String name) {
     return addressBook.get(name).getPhoneList();
    } }
```

History of gRPC

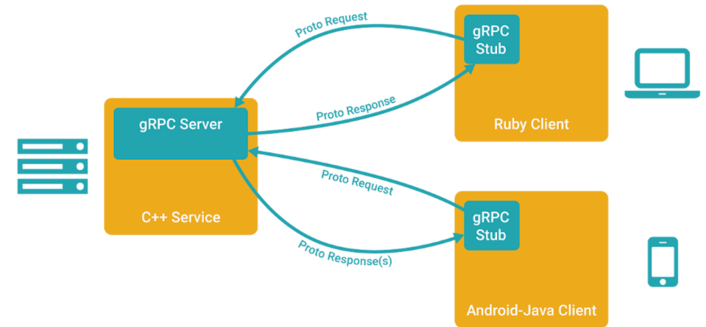
- Google open sourced in Feb 2015
- **Transport Protocol**: HTTP/2
- **Message format**: Protocol Buffers v3 (Binary)
- **Service definition**: Protocol Buffers v3 IDL
- Libraries in ~10 languages (native C, C#, Go, Java)
- RPC framework

gRPC client/server architecture



Getting Started

1. Define a service in a .proto file using Protocol Buffers: contains
   a) Message-Definition and
   b) Service-Definition.
2. Generate server and client stubs using the compiler
3. Extend the generated server class in your language to fill in the logic of your service
4. Invoke it using the generated client stubs



Message-Definition: Protocol Buffers v3

- A language-neutral, platform-neutral extensible mechanism for serializing structured data.
- Uses binary data representation with a Schema (smaller and faster than XML). Not human readable.
- When adding new fields, messages maintain backward compatibility.
- Code generators for many languages.
- Strongly typed.

```
syntax = “proto3”;

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

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phone = 4;
}
```

https://developers.google.com/protocol-buffers/docs/proto3

Message-Definition

- Strongly typed:
  - double, float
  - int32, int64
  - uint32, uint64
  - bool
  - string
  - bytes
  - enum
  - maps
- Default values, nested types
- “Repeated”-keyword for lists

```
syntax = “proto3”;

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

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    string number = 1;
    PhoneType type = 2;
  }

  repeated PhoneNumber phone = 4;
}
```

https://developers.google.com/protocol-buffers/docs/proto3

Generated Message Classes

```
class Person {
// required string name = 1;
public boolean hasName();
public String getName();

// optional int32 id = 2;
public boolean hasId();
public int getId();

// optional string email = 3;
public boolean hasEmail();
public String getEmail();

public static enum PhoneType {
  MOBILE(0, 0),  HOME(1, 1), WORK(2, 2)}

// repeated Person.PhoneNumber phone = 4;
public List<PhoneNumber> getPhoneList();
public int getPhoneCount();
public PhoneNumber getPhone(int index);
}
```

```
syntax = “proto3”;

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

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

  repeated PhoneNumber phone = 4;
}
```

Generated Builder Classes

- Person.Builder has the same getters plus the setters.
- Example:

```
Person john =
  Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .addPhone(
      Person.PhoneNumber.newBuilder()
        .setNumber("555-4321")
        .setType(Person.PhoneType.HOME))
    .build();
```

```
syntax = "proto3";

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

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

  repeated PhoneNumber phone = 4;
}
```

https://developers.google.com/protocol-buffers/docs/javatutorial

Service Definition

```
class AddressBook {
	Map<String, Person> addressBook;
	public Person findPerson(String name) {
	  return addressBook.get(name);
	}
	public List<PhoneNumber> listPhoneNumbers(String name) {
	  return addressBook.get(name).getPhoneList();
	}}
```

Server Code

Service Definition

```
class AddressBook {
        Map<String, Person> addressBook;
        public Person findPerson(String name) {
          return addressBook.get(name);
        }
        public List<PhoneNumber> listPhoneNumbers(String name) {
          return addressBook.get(name).getPhoneList();
        }}
```

Server Code

```
service AddressBook {
  rpc GetPerson(PersonRequest) returns (Person);
  rpc ListPhoneNumbers(PersonRequest) returns (stream Person)
}

message PersonRequest {
  string name = 1;
}

message Person{
            … // see Slide 7
}
```

Service-Definition

Generated Classes from Service Definition

- gRPC generates from the service def.:

1. Message objects (datastructures):
   PersonRequest.java and **Person.java**

2. Server-stubs (To be implemented):
   AddressBookGrpc
   .AddressBookImplBase.java

3. Client-stubs (callable):
   AddressBookGrpc.java

```
service AddressBook {
  rpc GetPerson(PersonRequest)
    returns (Person);

  rpc ListPhoneNumbers(PersonRequest)
    returns (stream PhoneNumbers)
}

message PersonRequest {
  string name = 1;
}

message Person{
   … // see Slide 7
}
```

Generated Classes from Service Definition

- **The client-API (callable methods)**:

```
public class AddressBookGrpc  {
   public void getPerson (…) {…}
   public void listPhoneNumbers (…) {…}

   public static AddressBookStub newStub (Channel channel) {}
   public static AddressBookBlockingStub newBlockingStub (Channel channel) {}
}
```

- **The server-stubs (abstract class to be implemented)**:

```
public static abstract class AddressBookImplBase implements io.grpc.BindableService {
   public void getPerson (…)
   public void listPhoneNumbers(...)
}
```

RPC

- `rpc GetPerson(PersonRequest) returns (Person)`

Client Code

```
// Access server
AddressBookBlockingStub server;
PersonRequest request = PersonRequest.newBuilder().setName("Patrick").build();
Person person = server.getPerson(request);
```

RPC

- `rpc GetPerson(PersonRequest) returns (Person)`

Client Code

```
// Access server
AddressBookBlockingStub server;
PersonRequest request = PersonRequest.newBuilder().setName("Patrick").build();
Person person = server.getPerson(request);
```

Server Code

```
class AddressBook {
        Map<String, Person> addressBook;
        public Person findPerson(String name) {
         return addressBook.get(name);
        }
        class AddressBookImpl extends AddressBookGrpc. AddressBookImplBase {
        @Override
        public void getPerson(PersonRequest request, StreamObserver<Person> observer) {
           observer.onNext(findPerson (request.getName()));
           observer.onCompleted();
}}}
```

Unary RPCs vs Streaming RPCs

- Unary RPCs:
  - the client sends a request to the server, then the server sends a response.

```
rpc GetPerson(PersonRequest) returns (Person)
```

- Streaming RPC:
  - the client and server may each send one or more messages.
  - messages sent on a stream are delivered FIFO.

```
rpc ListPhoneNumbers(PersonRequest) returns (stream Person)
```

Streaming RPC

```
rpc ListPhoneNumbers(PersonRequest) returns (stream Person)
```

Client Code

```
AddressBookBlockingStub server;
PersonRequest req = PersonRequest.newBuilder().setName("Patrick").build();
Iterator<Person.PhoneNumbers>  numbers = server.listPhoneNumbers(req);
```

Streaming RPC

```
rpc ListPhoneNumbers(PersonRequest) returns (stream Person)
```

Client Code

```
AddressBookBlockingStub server;
PersonRequest req = PersonRequest.newBuilder().setName("Patrick").build();
Iterator<Person.PhoneNumbers>  numbers = server.listPhoneNumbers(req);
```

Server Code

```
class AddressBook {
        public List<PhoneNumber> listPhoneNumbers(String name) {
          return addressBook.get(name).getPhoneList();
        }
        class AddressBookImpl extends AddressBookGrpc. AddressBookImplBase {
        @Override
        public void listAddresses(PersonRequest request, StreamObserver<Person> observer) {
            for (PhoneNumbers number : listPhoneNumbers(request.getName())) {
                responseObserver.onNext(number);
            }
            responseObserver.onCompleted();
} } }
```

An RPC ends when…

- both sides are done sending messages.
- either side disconnects.
- the RPC is cancelled or times out.

Synchronous vs. asynchronous

- Blocking / synchronous RPC calls
  - block until a response arrives from the server or raise an exception.

- Non-blocking / asynchronous RPC calls:
  - the response is returned asynchronously.

References

- https://github.com/grpc
- https://github.com/jonog/talks/blob/master/src/grpc/grpc-presentation.md
- http://platformlab.stanford.edu/Seminar%20Talks/gRPC.pdf
- http://www.grpc.io/docs/guides/concepts.html

- https://developers.google.com/protocol-buffers/docs/overview
- https://developers.google.com/protocol-buffers/docs/proto3