SERIALIZATION PROTOCOLS IN SCALA

a shootout

REASONING

so why is this even important?

SYSTEM DESIGNS ARE CHANGING

1. Microservice architectures are becoming mainstream
2. Queues (and distributed commit logs used as queues) show up in mainstream architectures
3. CQRS / ES is a common design nowadays, especially with the rising popularity of DDD
4. We're not using databases as integration layer anymore. (I hope?!...)

Distributed systems are about protocols, not implementations. Forget languages, protocols are everything.

@TimPerret

EVOLVABILITY

The ability of a system and its data to adopt to change

EVOLVABILITY

In monolithic systems with relational databases, a change in the data model was usually implemented by

- a schema migration
- code changes
- one big deployment

EVOLVABILITY EXAMPLES

- monolithic systems and relational databases
- replicated systems
- distributed systems in your Datacenter
- systems with code distributed to the User

NEW PERSISTENCE REQUIREMENTS

- on-disk formats behave significantly different than in-memory formats
- Data needs to be encoded/serialized/marshalled
- to get it back you need to decode/deserialize /unmarshall

SO HOW DO CAN WE DEAL WITH THIS?

USUAL SUSPECTS

PROGRAMMING LANGUAGE PROVIDED SERIALIZATION

we get `scala.Serializable` for free, so why not just use it?

NOOOOOOOOOOOO

- We are bound to one implementation language (but we're Java Compatible)
- The performance is awful
- It has some serious drawbacks:

Implement `Serializable` judiciously

[Joshua Bloch, Effective Java]

- decreases the flexibility to change the class once it is released
- The inner implementation of your class becomes the API
- increases the likelihood of bugs and security holes (Stack Overflows)
- increases the testing burden

SOME NOTES ABOUT PERFORMANCE

protocol	create	ser	deser	total	size
protobuf	121	1173	719	1891	239
thrift	95	1455	731	2186	349
kryo-flat	55	705	909	1614	268
avro-generic	329	1721	984	2704	221
scala/java-built-in	514	8280	36105	44385	1293

Source: jvm-serializers wiki

SOME NOTES ABOUT PERFORMANCE

- as long as its one of our four, it's probably fast enough
- if you need it faster, you'll have to do your own research
- just don't use the built in serializer

BUT... HOW ABOUT JSON? OR XML?

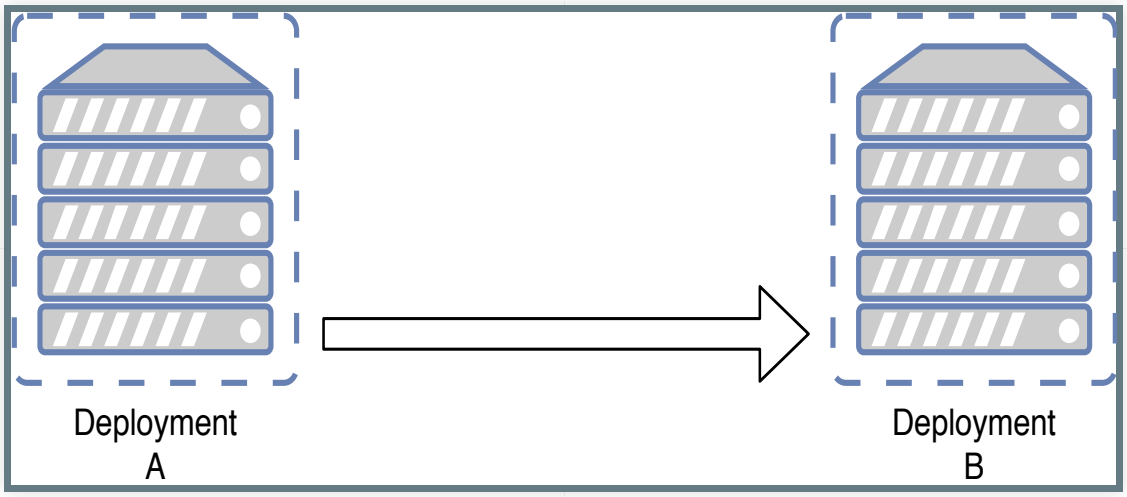
- widespread
- somewhat human readable
- still slow (depending on the benchmark)
- verbose
- number handling is difficult
- character encoding is difficult

BUT... HOW ABOUT JSON? OR XML?

There also exists a various range of schema extensions for JSON, like BSON and JSON Schema. Unfortunately they don't provide schema evolvability, so I'll skip them

SCHEMA EVOLVABILITY

READING DATA BETWEEN SCHEMAS



BACKWARDS COMPATIBILITY

This means a newer Schema can read Data encoded with an older Schema.

That's halfway (not) easy. You know how the previous schema was, you can deal with missing things in your code. If you are aware of that.

FORWARDS COMPATIBILITY

Reading Data from a Schema that is newer that the Schema of the running Software.

Ouch. How do you deal with things you don't know about?

FORWARDS COMPATIBILITY

Let a Protocol handle this for us!

PROTOBUF

What is Protbuf (protocol buffers)?

> *Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.*

Besides the specification, Google provides a compiler for Java

SCALAPB

ScalaPB is a protocol buffer compiler (`protoc`) plugin for Scala. It will generate Scala case classes, parsers and serializers for your protocol buffers

SCHEMA SAMPLE

```
syntax = "proto3";

package de.christianuhl.proto;

message Person {
    int32 user_id = 1;
    string firstname = 2;
    string lastname = 3;
}
```

USAGE WITH SBT

(live Sample)

LET'S CHECK THE ENCODING ON DISK

field tag	type	value	length	bytes / value
00010	010	12	09	43687269 73746961 6E
00011	010	1A	03	0355686C

PROTO 2 VS PROTO 3

Significant changes:

- No presence logic anymore
- All values are optional by default, `required` semantic is removed
- `Map` Type
- Timevalues

WHY THE RADICAL 'EVERYTHING IS OPTIONAL'?

- `required` breaks schema evolvability anyways
- loads of production issues
- was already banned in Google
- `optional` became redundant and was removed as well

THRIFT

WHAT IS THRIFT?

... is an interface definition language and binary communication protocol that is used to define and create services for numerous languages. It is used as a remote procedure call (RPC) framework

wikipedia

WHAT IS THRIFT?

- it has large conceptual similarities to protobuf
- originally developed at Facebook
- re-open-sourced later as fbthrift

SCALA & SBT INTEGRATION

Twitter supplies scrooge

Scrooge

SCHEMA SAMPLE

```
struct Person {
    1: optional i64     userId,
    2: required string  firstname,
    3: required string  lastname
}
```

USAGE WITH SBT

(live Sample)

LET'S CHECK THE ENCODING ON DISK

Type	Field Tag	Length	bytes	value
0B	0002	00000009	4368726973746961 6E	Christian
0B	0003	00000003	55686C	Uhl

WHAT MAKES THRIFT SPECIAL?

- It's more than just serialization, it's a whole RPC framework
- It embraces a whole family of encodings (They call it protocols)
- Scala ecosystem only gets contributions by Twitter
- otherwise, it's remarkably similar to protobuf

AVRO

differs from Protobuf and Thrift:

- Reader and Writer Schema can be different, Data brings its own schema
- less type information for the payload needed
- no manually assigned field ids

ALL BOILS DOWN TO HAVING DIFFERENT SCHEMAS FOR READING AND WRITING

SCHEMA SAMPLE

```
record Person {
    union {null, long} user_id = null;
    string firstname;
    string lastname;
}
```

USAGE WITH SBT

using avrohugger and avro4s

- avro4s
- avrohugger

(live Sample)

LET'S CHECK THE ENCODING ON DISK

first, 248 Bytes of schema

lenght	sing	value	bytes	value
0001001	0	12	43687269 73746961 6E	Christian
0000011	0	06	55686C	Uhl

SO HOW DO WRITER'S AND READER'S SCHEMA GO TOGETHER?

- they need to be *compatible*, but not equal
- resolution rules are in the Avro specification
- adding or removing only of fields with default values

HOW DOES DATA AND SCHEMA FIND TOGETHER?

1. You include it in the Payload (as in the Sample)
2. You versionize it, store it yourself and join it later (typical Database Application)
3. Roll your Schema Registry and do Avro RPC

WHAT BENEFITS TO WE GET FROM THAT?

1. Support for dynamically generated schemas (Protobuf and Thrift want to be hand-crafted)
2. Simpler interoperability with dynamically typed languages

AND THE DOWNSIDES

1. Protocol Overhead or Runtime Dependency, pick a poison
2. Need to verify schema changes are compatible

KRYO

THE 'ODD' ONE OF OUR LITTLE LIST

- Object Graph serialization Framework for the JVM
- No Schema evolution
- Blazing fast JVM-to-JVM communication
- good candidate for live cluster communication (Akka e.g.)

Source

USAGE WITH SBT

using twitter chill

- chill

(live Sample)

LET'S CHECK THE ENCODING ON DISK

16 bytes, maximum density

field	type(?)	value (last byte starts with 1)	actual
01	01	4368 72697374 6961EE	Christian
02	01	5568 EC	Uhl

KRYO SUMMARY

- no Schema
- therefore no Schema Evolution
- Good for Message-Serialization for running instances
- you do not want to persist these serialized Values

SUMMARY

SUMMARY

Framework	Notes
Protobuf	nice integration, clear concepts,confusing 2->3 version change
Thrift	clear concepts,bigger framework, lacking integration
Avro	good integration, good for big files,embedded schema concept
kryo	speed, simplicity,no evolvability, not for persisting

RELEVANT BOOKS

- Effective Java
- Desinging Data Intensive Applications

QUESTIONS?

My twitter is @chrisuhl