Dealing with performance challenges

Optimized Data Formats

Sastry Malladi

eBay, Inc.

Agenda

- API platform challenges
- Performance : Different data formats comparison
- Versioning
- Summary

Fun facts about eBay

- eBay manages …
  - **Over 97 million active users**
  - **Over 2 Billion photos**

- eBay users worldwide trade on average $2000 in goods every second ($ 62 B in 2010)
- eBay averages 4 billion page views per day
- eBay has over 250 million items for sale in over 50,000 categories
- eBay site stores over 5 Petabytes of data
- eBay Analytics Infrastructure processes 80+ PB of data per day
- **eBay handles 40 billion API calls per month**

In 40+ countries, in 20+ languages, 24x7x365

>100 Billion SQL executions/day!

APIs / Services @eBay

- It's a journey !
- History
  - One of the first to expose APIs /Services
  - In early 2007, embarked on service orienting our entire ecommerce platform, whether the functionality is internal or external
  - Support REST + SOA
  - Have close to 300 services now and more on the way
  - Early adopters of SOA governance automation
- Technology stack
  - Mix of highly optimized home grown + best of breed open source components , integrated together – code named Turmeric
  - Open sourced @ http://ebayopensource.org

Types of APIs

- SOA
  - Formal Contract, interface (WSDL or other)
  - Transport / Protocol agnostic (bindings)
  - Arbitrary set of operations
  - Code generation is typically always involved
  - Meant for sophisticated application developers
- REST
  - Based on Roy Fielding's dissertation
  - Web/Resource oriented
  - Suits well for web based interactions
  - Piggy backs on HTTP verbs : GET, POST, PUT, DELETE
  - No formal contract
  - Hypermedia / Discoverability /Navigability
  - Ease of use

Most external APIs tend to be REST based for ease of use and simplicity

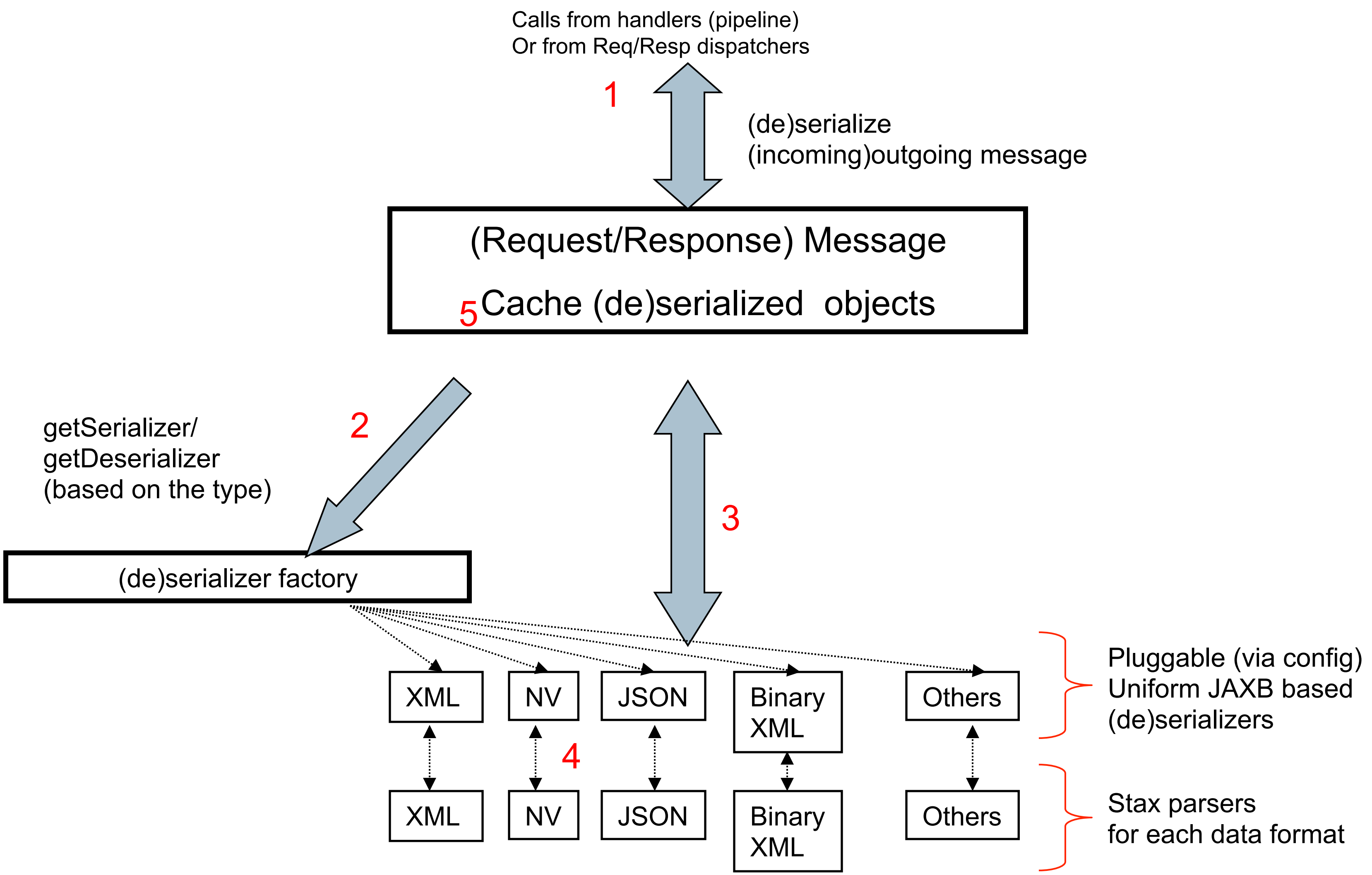
Data formats

- The Web API request/response messages have to exchange messages in commonly understandable data formats, independent of the programming language. XML, JSON are two of the most popular formats.
- Over the years, these data formats continued to evolve and more formats are popping up every now and then, each one claiming to have its own advantages.
- When the API is exchanging messages with external clients, interoperability and ease of use are very important and hence you would commonly use JSON/XML.
- But when exchanging messages with internal clients, it may support additional optimal formats, for performance reasons.
- How do we support these evolving formats, without having to require clients/servers to rewrite their code. Turmeric framework and provides this architecture and support many data formats out of the box.
- There is a cost to serialize and deserialize objects (in whatever language your client/server is implemented) into these wire data formats.
- The question is, how do we reduce this cost ? What is the best format to use in what circumstances ?

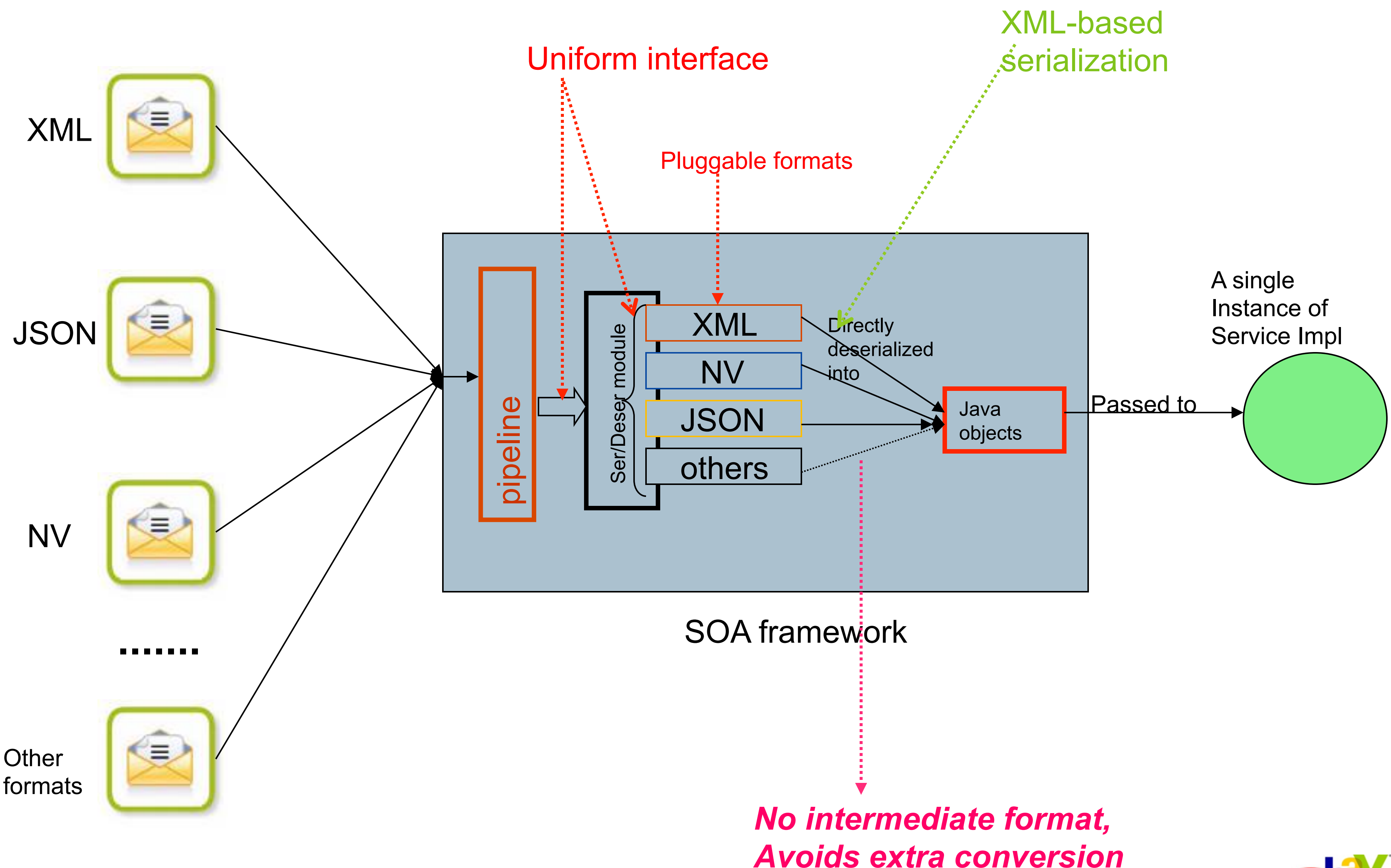
API platform and design challenges

- API Platform challenges
  - Performance : Serialization / Deserialization cost
  - Data formats evolution
  - Versioning
  - Hypermedia support
  - Providing/generating documentation
  - Security
- API design challenges
  - Ease of use
  - Interoperability
  - Backward compatibility
  - Granularity

Turmeric : Pluggable Data Formats Using JAXB



Turmeric : Native and uniform (de)serialization



Agenda

- API platform challenges
- Performance : Different data formats comparison
- Versioning
- Summary

Performance Challenges

- The solution to plugin different data formats (XML, JSON, NV, FastInfoset) seamlessly under JAXB works great.
- However, with these formats, we observed latency issues
  - For large payloads and high volume environments, serialization and deserialization cost is significant and not acceptable
  - Size of the serialized message also is significant leading to network bandwidth costs
- Alternatives
  - Looked at true binary formats like Protobuf, Avro and Thrift
  - They looked very promising in terms of serialization and deserialization times

Challenges with the alternative formats

- Each of these formats have their own schema/IDL to express the message definitions
- Not every format supports all the schema types and structures.
- They each have a codegen mechanism that generates corresponding bean classes, which are NOT necessarily compatible with any existing classes
- Testing : Simulating a given message sized structure uniformly across all formats isn't trivial

```
Note : BTW, there are some existing benchmarks for comparing some of these formats on the web (http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking) - But these benchmarks don't test different payload structures and sizes
```

Formats tested

- XML
- JSON (various implementations – Jackson, Jettison, Gson)
- FastInfoSet
- Protobuf
- Protostuff
- Avro
- Thrift
- MessagePack

Areas of comparison

- Serialization / Deserialization cost
- Network bandwidth (serialized message size)
- Schema richness (support for types that we need)
- Versioning
- Ease of use
- Backward/Forward compatibility
- Interoperability
- Stability / Maturity
- Out of the box language support
- Data format evolution – Velocity of changes

Benchmark context

- Goal
  - Understand the best optimized formats for reduced serialization/deserialization/ bandwidth (size) cost
  - Understand the overall best format to use, considering other factors like ease of use, versioning, schema richness, stability, maturity, etc.
- Non-goal
  - Each of these formats have their own RPC mechanism, and it is not our goal to evaluate or use that.
- Benchmark
  - Simulated Message structure, tailored to the desired size
    - With 4 levels of nested tree structure (configurable), containing all representative types
    - Randomness introduced, to simulate distinct data for each message instance
  - Environment
    - Everything in the same JVM, so pure serialization/deserialization time – no network cost
    - MacBook Pro : OS : 10.6.7, Java 6
      - 2.66 GHz i7 processor, 8GB RAM

Note : Everything here needs to be taken as relative numbers – don't pay too much attention to the absolute numbers

How they compare - Functionally

Protobuf	Avro	Thrift
➢ Own IDL/schema ➢ Sequence numbers for each element ➢ Compact binary representation on the wire ➢ Most XML schema elements are mappable to equivalents, except polymorphic constructs, enums, choice etc. ➢ Inheritance through composition ➢ No attachment support ➢ Versioning is similar to XML, a bit more complex in implementing due to sequence numbers ➢ Originally from Google, has been around for a while – current version – 2.4 ➢ Available (officially) in Java, C ++, Python	➢ JSON based Schema ➢ Schema prepended to the message on the wire (dynamic typing) ➢ Supports dynamic as well as static typing ➢ Compact binary representation on the wire ➢ Most XML schema elements are mappable to equivalent, except polymorphic constructs. Work around exists for tree like structures ➢ Inheritance through composition ➢ No attachment support ➢ Versioning is easier ➢ Originally developed as part of the Apache Hadoop Family, current version 1.5 ➢ Available in C, C++, C#, Java, Python, Ruby, PHP	➢ Own IDL/schema ➢ Sequence numbers for each element ➢ Compact binary representation on the wire ➢ Most XML schema elements are mappable to equivalents, except polymorphic constructs and tree like structures ➢ Inheritance through composition ➢ No attachment support ➢ Versioning is similar to XML, a bit more complex in implementing due to sequence numbers ➢ Originated by Facebook – curent release 0.7.0, but has been around for a while ➢ Available in pretty much all languages

How they compare - Functionally (contd.)

Protostuff	MessagePack	FastInfoset
➢ Everything is same as protobuf, with additional features like streaming and support for existing pojos ➢ Done by some individual committer ➢ Version 1 ➢ Can write to JSON/XML formats	➢ Has no schema ➢ Compact binary representation on the wire ➢ No code generation ➢ All fields in the message needs to be public (in java) ➢ No tree like structures ➢ No attachment support ➢ Not much support for versioning ➢ Available in c/c++, Ruby, Python, Perl, Node.JS ➢ Started by an individual, relatively recent	➢ XML schema ➢ Everything same as XML, except that the representation on the wire is semi-binary ➢ Based on ISO/ITU standard using ASN.1 notation

Message structure (equivalent in different formats)

```
<complexType name="XMLMessage" ">
     <sequence>
          <element name="integer" type="xsd:int" minOccurs="1" maxOccurs="1" />
          <element name="astring" type="xsd:string" minOccurs="1" maxOccurs="1" />
          <element name="adouble" type="xsd:double" minOccurs="1" maxOccurs="1" />
          <element name="strings" type="xsd:string" minOccurs="1" maxOccurs="Unbouned" />
          <element name="selfRef" type="tns:XMLMessage" minOccurs="1" maxOccurs="1" />
          <element name="selfRefList" type="tns:XMLMessage" minOccurs="1"
maxOccurs="Unbounded" />
      </sequence>
</complexType>
```

XML

```
message ProtobufMessage {
    optional int32 integer = 1;
    optional string astring = 2;
    optional double adouble = 3;
    repeated string strings = 4;
    optional ProtobufMessage selfRef = 5;
    repeated ProtobufMessage selfRefList = 6;

}
```

Protobuf

Message structure (equivalent in different formats) – contd.

```
"types" : [
    {
       "type" : "record",
       "name" : "AvroMessage",
       "fields" : [
           {"name" : "integer", "type" : "int" },
           {"name" : "astring", "type" : "string" },
           {"name" : "adouble", "type" : "double" },
           {"name" : "strings", "type": [{"type": "array", "items": "string"}, "null"] },
           {"name" : "selfRef", "type" : ["AvroMessage", "null"]},
           {"name" : "selfReflist", "type" : [{"type": "array", "items":
"AvroMessage"},"null"]}
        ]
     }
   ]
```

Avro

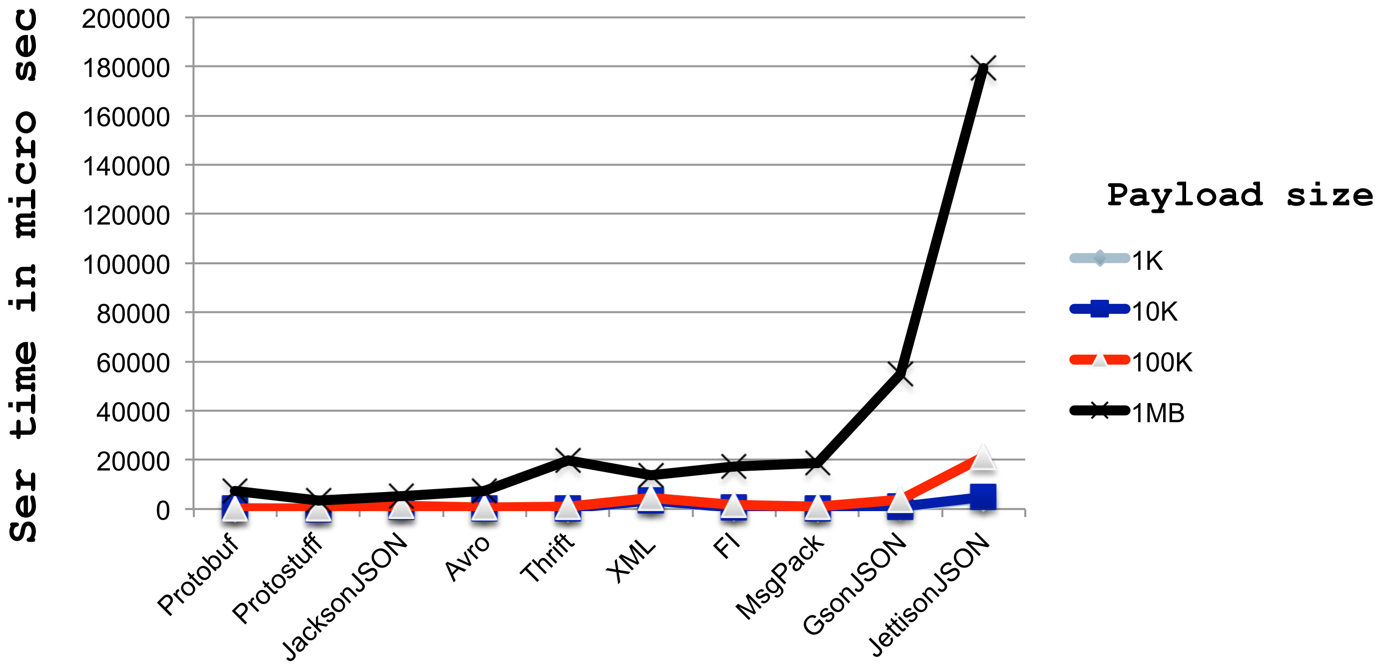
```
struct ThriftMessage2 {
 1:   optional i32 integer,
 2:   optional string astring,
 3:   optional double adouble,
 4:   list<string> strings,
}

struct ThriftMessage {
 1:   optional i32 integer,
 2:   optional string astring,
 3:   optional double adouble,
 4:   list<string> strings,
 4:   optional ThriftMessage2 selfRef,
 5:   optional list<ThriftMessage2t> selfRefList,
}
```

Thrift

Benchmark runs

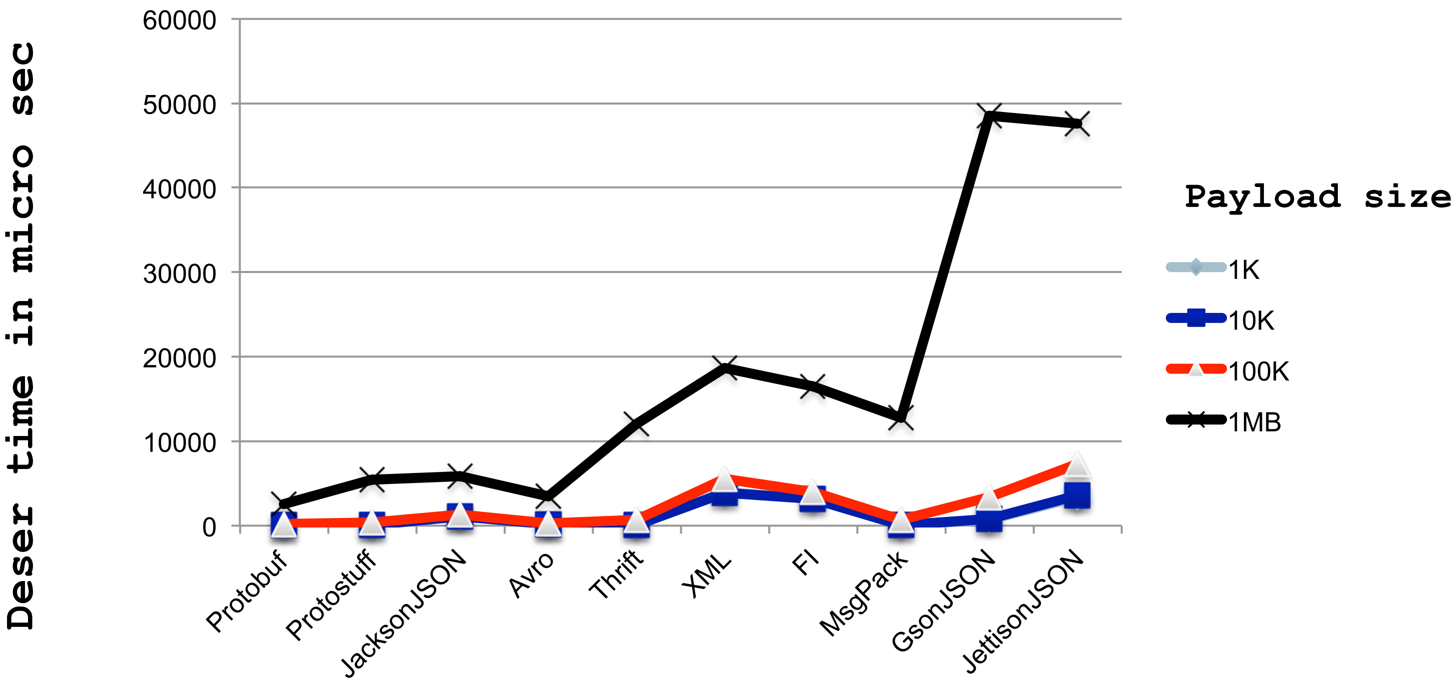
- Each data format test is run in a separate JVM instance
- Each test has 1000 iterations
- Each payload size run is also done in a different JVM.
- The message content is random for each instance, to simulate real world payloads.
- 95th percentile average is measured.

Serialization time – 95th percentile



- At lower payload sizes (up to 100K)
  - protobuf, protostuff, MsgPack and Avro are the best in that order and are comparable.
- At higher payload sizes (1MB)
  - Protostuff is best, followed by JacksonJSON, protobuf and Avro
  - Avro and Protobuf are more or less the same
  - JacksonJSON, while worse than protobuf at smaller payloads, is better at higher payloads.
  - JettisonJSON and GsonJson are out of whack

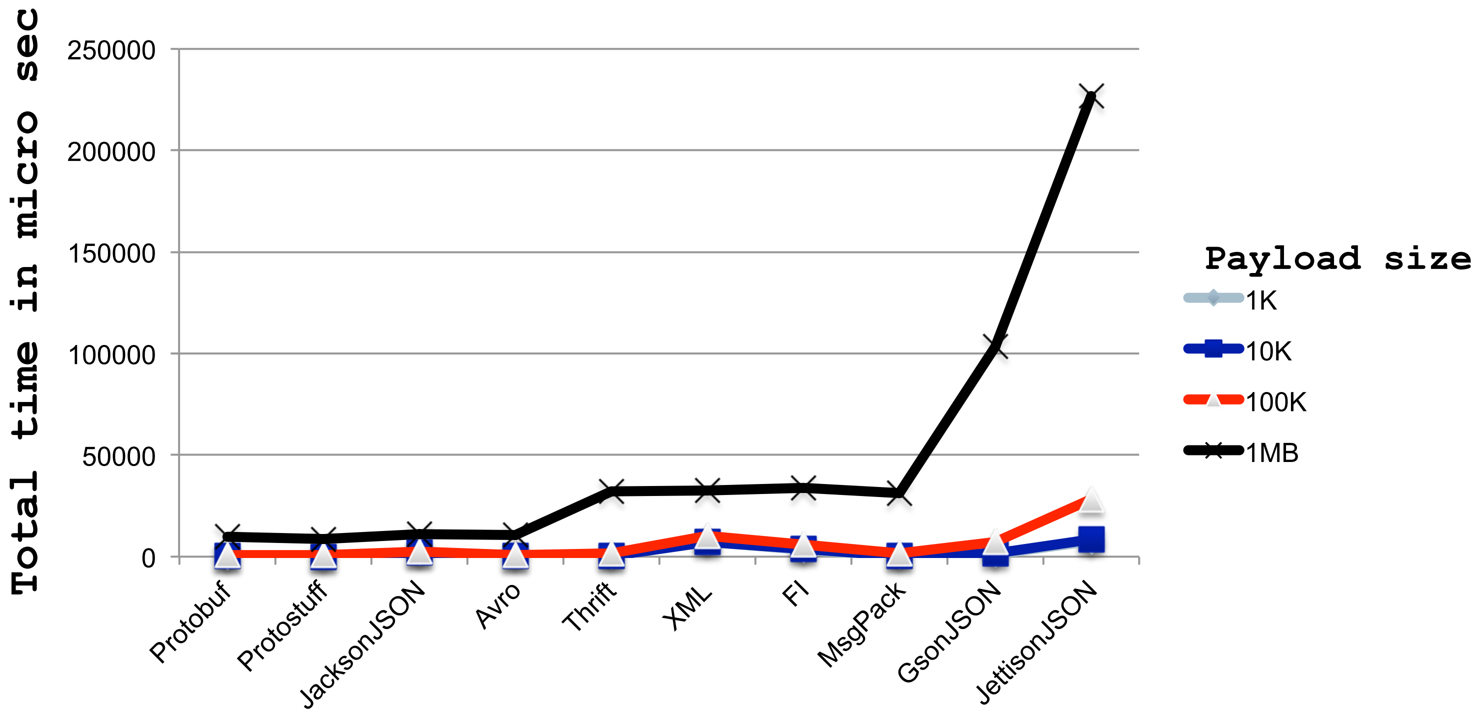
Deserialization time – 95th percentile



- For deserialization, protobuf is the best of all, followed by Avro, Protostuff and JacksonJSON

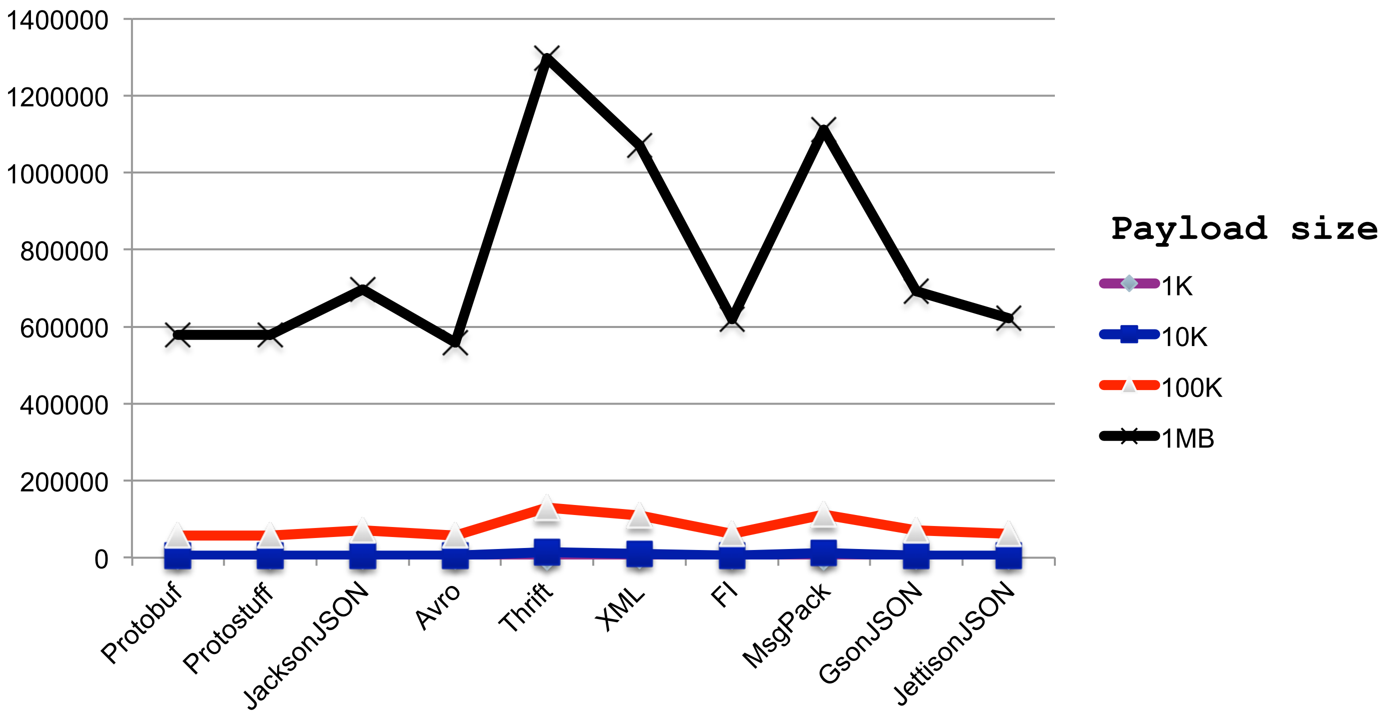
- Thrift and MsgPack, while good at lower payloads, deteriorate at higher payloads.

Total Time – 95th percentile



- Overall, for higher payloads, best formats :
  - Protostuff, protobuf, Avro and JacksonJSON in that order
- Overall, for lower payloads, best formats :
  - Protostuff, protobuf, Thrift and Avro

Serialized Payload size



- XML, Thrift and MsgPack don't seem to have any edge, i.e., no reduction in size
- All other formats have reduced serialized size that vary between 30-40% reduction gain.

Here is what you have been waiting for …

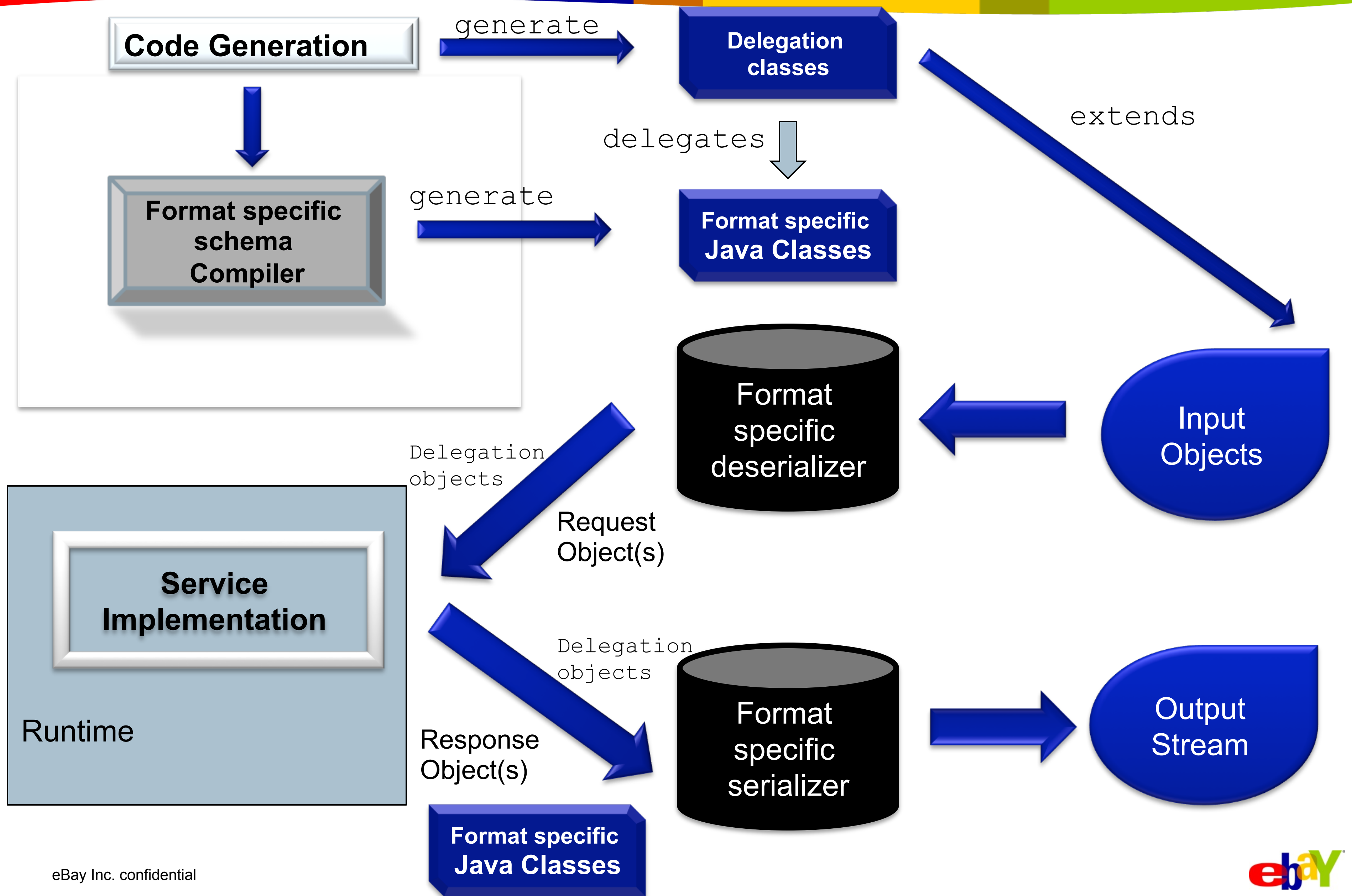
Our benchmark results indicate that …

- Considering all the factors (performance, interoperability, schema richness, usability) Jackson JSON is overall the best one to use

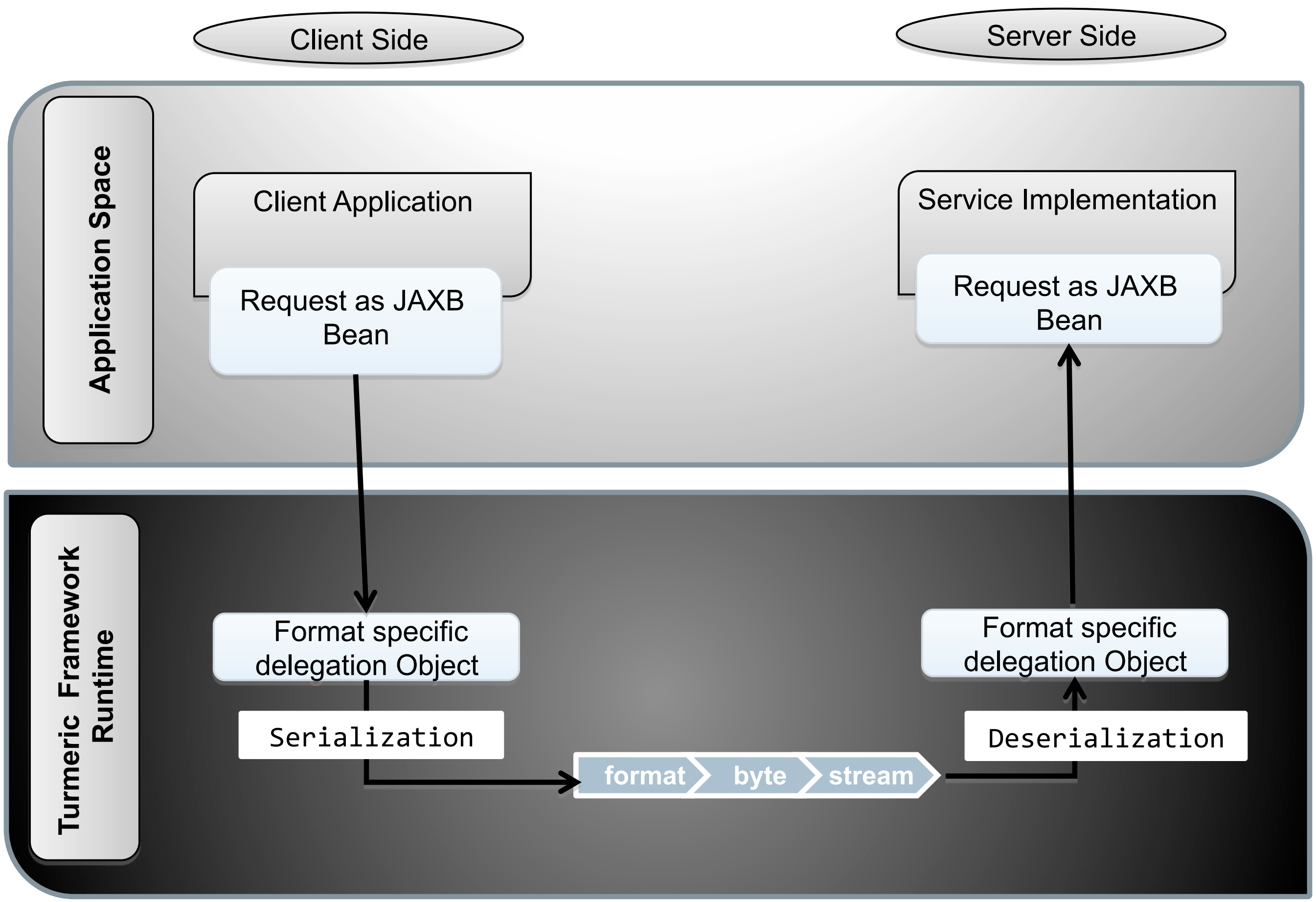
- But if performance is an absolute must, and can compromise on the ease of use and schema limitations, interoperability, then Protostuff is the best one to use.

So how did we leverage this ?

General (de) serialization flow

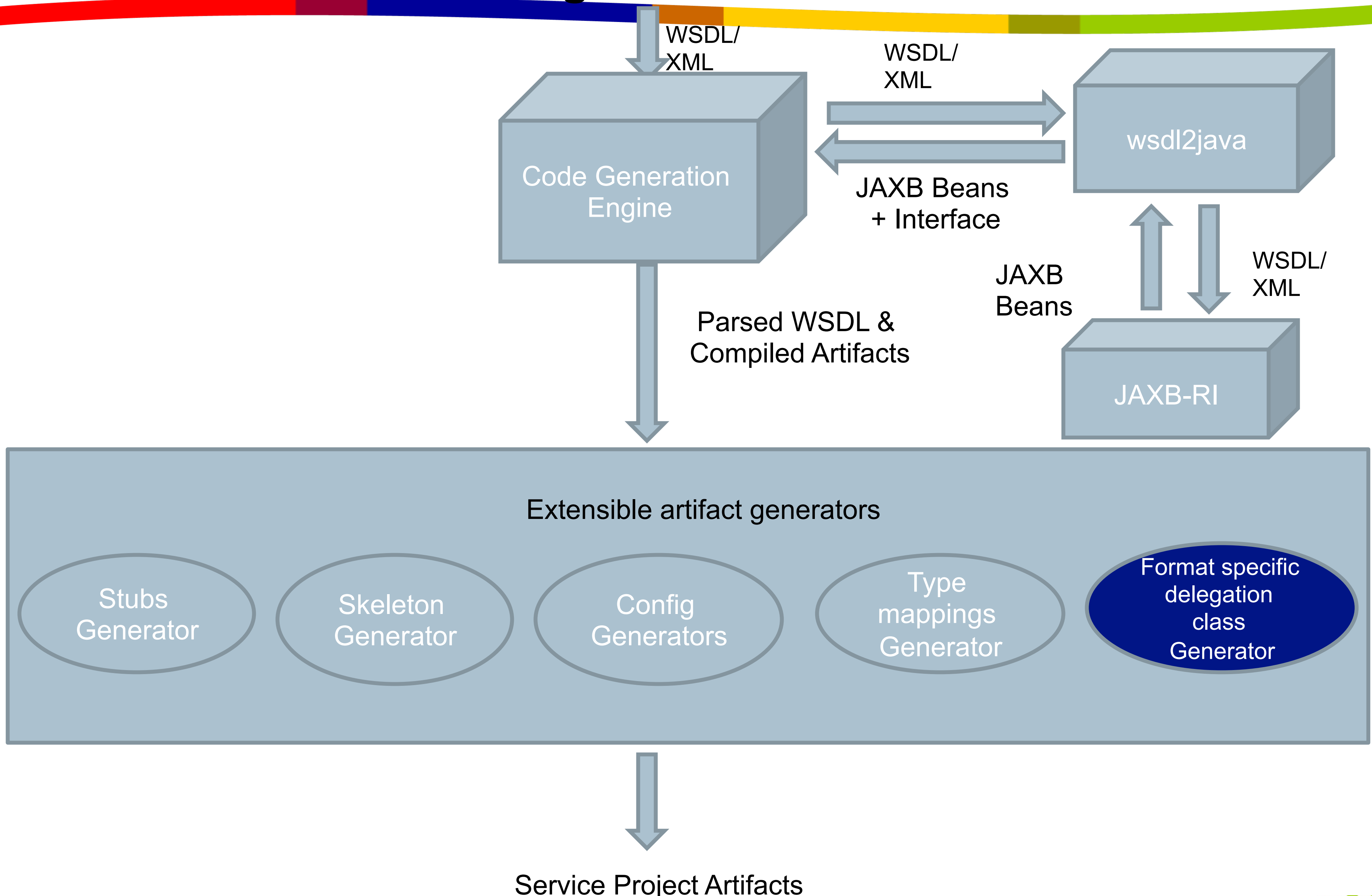


Keeping the same existing interface – Message schema expressed in XML



Respective Message schemas can be queried using “?proto”, “?avro” etc.

Turmeric : Pluggable data format specific artifact generators



Restful API

- The same concept of plugging in different data formats (media types) is done for restful APIs
- JAX-RS specification allows plugging in different media type providers
  - MessageBodyReader (Deserializer) and MessageBodyWriter (Serializer)
- Content negotiation can be done using the standard HTTP headers
- A small demo of hypermedia and a good rest API (time permitting)

How to use this and leverage protobuf, for example ?

- Get Turmeric
  https://www.ebayopensource.org/index.php/Turmeric/HomePage
- Generate service and client with Turmeric Eclipse Plugin
  - If compatible, all protoc and adapter classes are generated automatically
- Implement the service and client application code as usual
- At runtime, set request/response header format to use Protocol Buffers
- You are all set!

Agenda

- API platform challenges
- Performance : Different data formats comparison
- Versioning
- Summary

Versioning

- Versioning is a perennial problem in APIs / Services world
- Change is inevitable and therefore APIs (their requests and responses) do change from time to time
- There are no standards to do versioning
- The question is, what is the best approach, which is simple and understandable by the consumers ?
- We followed this convention
  - Version internally has 3 components : Major, Minor and Maintenance (e.g 1.2.1)
    - Maintenance version is bumped up for any bug fixes (no interface change)
    - Minor version is bumped up for any backward compatible interface changes
    - Major version is bumped up for any backward incompatible changes (or for major new functionality)
    - In any given major version, the latest minor version is always compatible with all the previous minor versions.
    - We have some semi-automated tools to enforce these guidelines

Versioning (contd.)

- But externally, we don’t want to expose all of that.
  - Version externally needs to see only component : Major (e.g. V1 or V2)
  - Standardized format
    - http[s]://svcs.ebay.com/<domain>/<service>/V? (versioning the domain) OR
    - http[s]://svcs.ebay.com//<domain>/V?/<service> (Versioning the service/resource)
  - e.g. : /finding/V?/Items?keyword=ipod

- Depending on which data format is used, implementation difficulty varies, as touched upon during the data format comparisons.
- Resource Versioning for Rest APIs follows similar pattern
  - http[s]://host:port/<domain>/V?/<Resource> or
  - https[s]://host:port/<domain>/<Resource>/V?
  - Versioning can also be negotiated using the Accept header (accept parameters)

Summary

- API platform itself has various challenges, in addition to the API design challenges
- Performance, Usability, Versioning and Interoperability are some of the key aspects to consider
- APIs are used both internally and externally, and the type of challenges vary between internal and external
- Binary data formats offer performance advantages, but bring along certain restrictions and challenges
- We have done a benchmark study to understand what formats are best under what circumstances and concluded that JSON (specifically jackson parser) is good for majority of the use cases and for high performance critical services, protostuf is the best
- Versioning is another major challenge that we dealt with simple conventions
- Some of the innovations we have done at eBay are open sourced under Turmeric project

Q & A

Thank you

smalladi@ebay.com

Back up slides

Serialization - 95th percentile data

	1K	10K	100K	1MB
Protobuf	79	86	435	7332
Protostuff	63	72	238	3288
JacksonJSON	944	862	1184	5249
Avro	396	340	485	7388
Thrift	77	137	1026	19875
XML	3340	3304	4545	13866
FI	432	487	1789	17222
MsgPack	60	105	898	18677
GsonJSON	647	802	3833	54948
JettisonJSON	4100	4808	21126	179256

Deserialization – 95th percentile data

	1K	10K	100K	1MB
Protobuf	51	52	200	2554
Protostuff	59	54	371	5398
JacksonJSON	1125	1051	1325	5872
Avro	251	217	285	3437
Thrift	96	92	640	12086
XML	4012	3927	5553	18631
FI	3126	3130	3983	16494
MsgPack	112	108	706	12764
GsonJSON	704	836	3332	48476
JettisonJSON	3358	3591	7302	47625

Total time – 95th percentile data

	1K	10K	100K	1MB
Protobuf	130	138	635	9886
Protostuff	122	126	609	8686
JacksonJSON	2069	1913	2509	11121
Avro	647	557	770	10825
Thrift	173	229	1666	31961
XML	7352	7231	10098	32497
FI	3558	3617	5772	33716
MsgPack	172	213	1604	31441
GsonJSON	1351	1638	7165	103424
JettisonJSON	7458	8399	28428	226881

Serialized Size – 95th percentile data

	1K	10K	100K	1MB
Protobuf	3106	5326	58655	578692
Protostuff	3105	5325	58656	578553
JacksonJSON	3739	6410	70536	695831
Avro	3003	5149	56658	558931
Thrift	1505	13956	130564	1296773
XML	5814	9929	108684	1071478
FI	3403	5780	62827	619291
MsgPack	1708	11962	111939	1112197
GsonJSON	3708	6365	70120	691815
JettisonJSON	3377	5768	63186	622847