Protobuff Serialization And Alternatives

Azat Satklyčov
azats@seznam.cz,
http://sahet.net,
https://github.com/asatklichov/protobuf_and_alternatives

Agenda

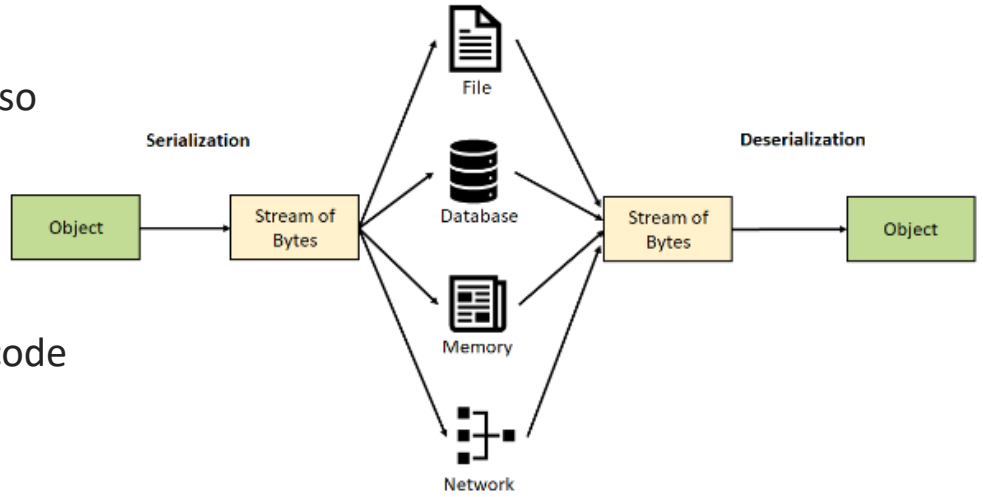
- Java Serialization Mechanism
- Protocol Buffers
- FlatBuffers
- Cap’n Proto
- MessagePack
- SPE
- Comparison Matrix
- Architectural Change

Java Serialization Mechanism

Java Serialization is a mechanism by which Java objects (in heap) can be converted into a byte stream (reside in disk, DB, etc.), and consequently, can be reverted back into a copy of the object(in heap). Implemented via using the java.io.Serializable (or Externalizable) interface and ObjectOutput#writeObject(), ObjectInputStream#readObject(), and in Java 9 ObjectInputFilter. Usage: persist objects, deep cloning, communication (RMI, Apache Hadoop, Spark, etc.) - ephemeral network traffic and long-term data storage

Complex

- Process is recursive – whole graph of objects included (heavy object)
- If superclass is not serializable no parent constructor called
- SerialVersionUID – InvalidClassException thrown if not same on both objects, so better explicitly declare it & keep incremented.
- Now work well if you share data with apps written in C++, Python



Security concerns

- deserialization from untrusted sources, e.g. readObject may have execution code with no instruction
- Deserialized object graph can consume huge memory, open to DoD attack
- Sensitive data exposures

Security Libraries

Apache Commons Lang SerializationUtils, & update Java for security patches

Serialization Alternatives and Future

Custom Java NIO ByteBuffer, XML, JSON, Protocol Buffers, Cap'n Proto, FlatBuffers, Message Pack, Apache Thrift, Avro, Kryo etc.

e.g. Application Messages

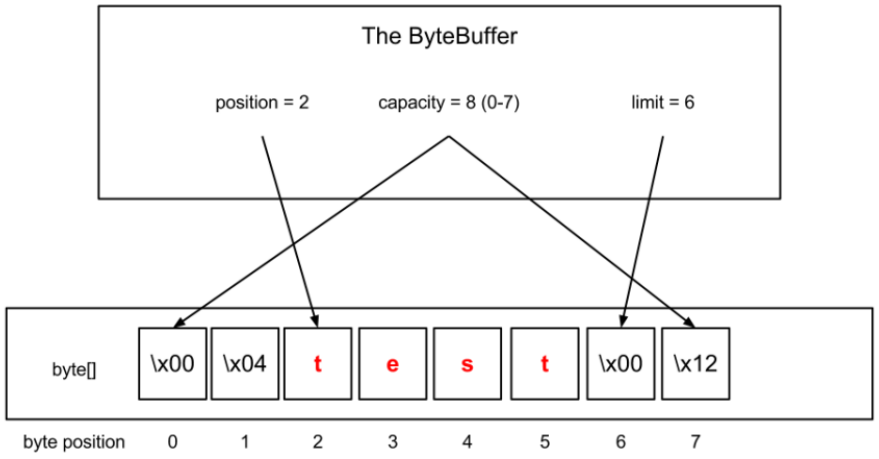
The Buffer classes are the foundation upon which Java NIO is built. Testing Tools uses ByteBuffer class to convert messages into bytes and to translate back.

- Pure Java code - no external library used
- Used as dependency for both ServerA and Server2
- Simple to code
- Good for fast low-level I/O to implement TCP/IP based apps.

```
@Override
protected void _extractMessage(ByteBuffer newbuffer)
throws Exception {
   findType = buffer.get();
   dataLength = buffer.getInt();
    address = new MFAddress(buffer);
    findRange = buffer.getInt();
    int lth = Numbers.getUShort(buffer.getShort());
    byte[] buf = new byte[lth];
    buffer.get(buf);
}
```

Limitations

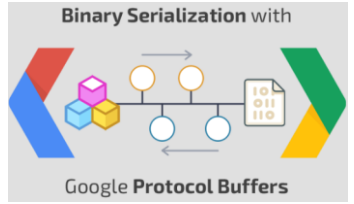
- Small change can lead broken application unless tested with caution
- Using ByteBuffer is a source of confusion
- Messages are not readable – working with bytes are not easy
- For complex objects writing to bytes and reading it from is not easy
- Happens not to “consume” a byte buffer when reading it
- Only buffers used without channel involvement (File or Socket)
- Messages only used by JAVA, JNI is an only option which is not a easy wa



Pain points:

- Non-extensible protocol ⇒ Changes to the protocol require perfectly coordinated upgrades of both the server and client components. Past attempts to make the changes backward compatible introduced new bugs.
- Tight coupling ⇒ it’s difficult to make even the slightest changes in the structure of these binary messages.

Protocol Buffers



Protobuf serializes data in lang-agnostic (JAVA, C++, JS, ..) way. E.g. gPRC, GoogleCloud, EnvoyProxy uses it.

- **Efficient** Data Compaction, Serialization and Deserialization. Simple to Use
- Supports extending format seamlessly to add new fields, delete existing fields without breaking existing services.
- The **format** is suitable for both ephemeral network traffic and long-term data storage.
- **Types**: bool, int32, float, double, string, **Fields**: optional and required (used in proto2), repeated
- **Simpler, much faster** for smaller messages < 1MB. Protobuf3 supports inheritance
- Classes generated by compiler (automatically) are Immutable (thread safe).
- Converting json-messages to protobuf is straightforward.
- Protocol Buffers are similar to the Apache Thrift, Ion, and Microsoft Bond protocols, offers gPRC
- **Cross-project** support, e.g. of proto definitions in Google are timestamp.proto and status.proto
- **Cross-language** compatibility: Different systems can have different lang. but can exchange data
- Integration with messaging systems, e.g. Kafka and others

Protocol Buffers

Developer(s)	Google
Initial release	Early 2001 (internal)[1] July 7, 2008 (public)
Stable release	25.2 / 9 January 2024; 10 days ago[2]
Repository	github.com/protocolbuffers/protobuf
Written in	C++, C#, Java, Python, JavaScript, Ruby, Go, PHP, Dart
Operating system	Any
Platform	Cross-platform
Type	serialization format and library, IDL compiler
License	BSD
Website	protobuf.dev

Characteristics or Limitations

- Protobuf2 to Protobuf3 – migration needed, e.g. tags, required fields (protobuf2), etc.
- Auto generated code **differentiates regarding different versions** (even not modified) e.g. v3.4.0 & v3.25.0
- Same data may have different binary serializations, U can't compare two messages unless you parse them
- only ASCII format
- Format is space efficient, messages are not compressed. library doesn't provide compression out of the box
- No schema: messages are serialized into a binary wire format which is compact, forward- and backward-compatible, but not self-describing [no way to tell the names, meaning, or full datatypes of fields without an external specification].

Protoscope is a very simple, human-editable lang. to represent and emit Protobuf wire format: ***Protobuf + Rotoscope***

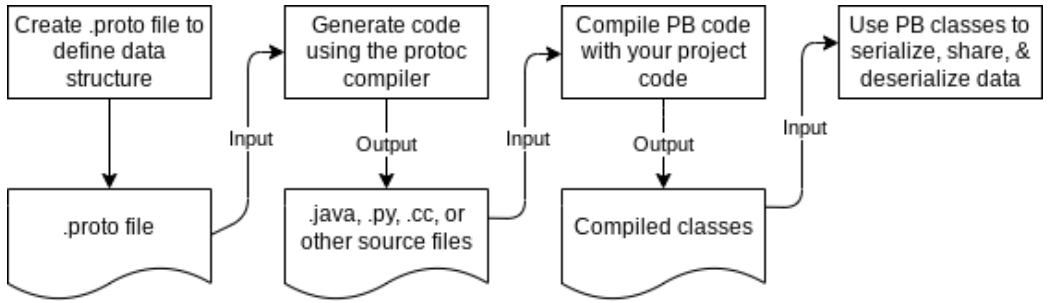
Steps how to use protobuf to generate messages, and for serialization and deserialization

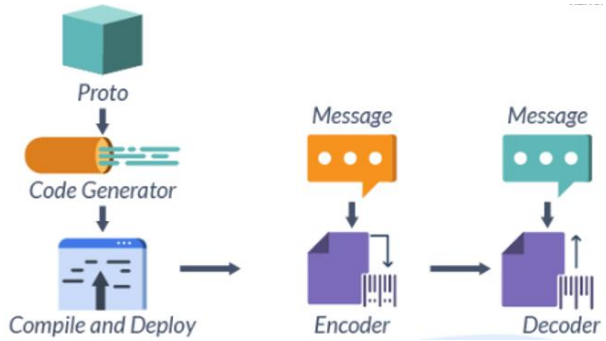
1. Download your OS relevant protobuf-compiler & set env-variable for it
2. Define maven dependencies and plugins, see <GitHub-link>
3. Define your data-structure (message) using protobuf format, e.g. ?.proto, for style guide look here
4. Generate[compile] messages (classes) from Protobuf file using 'protoc' compiler

```
>protoc -I=src/main/proto --java_out=src/main/java person.proto    //via command line
>mvn protobuf:compile -DprotocExecutable="C:/apps/protoc-25.2-win64/bin/protoc.exe"  //maven way
```

5. Start using generated classes (**immutable** implements automatic encoding&parsing of protocol buffe with an efficient data format, also provides util-chained methods) in your app. to exchange messages

```
C:\Users\as892333>protoc --version
libprotoc 25.2
```





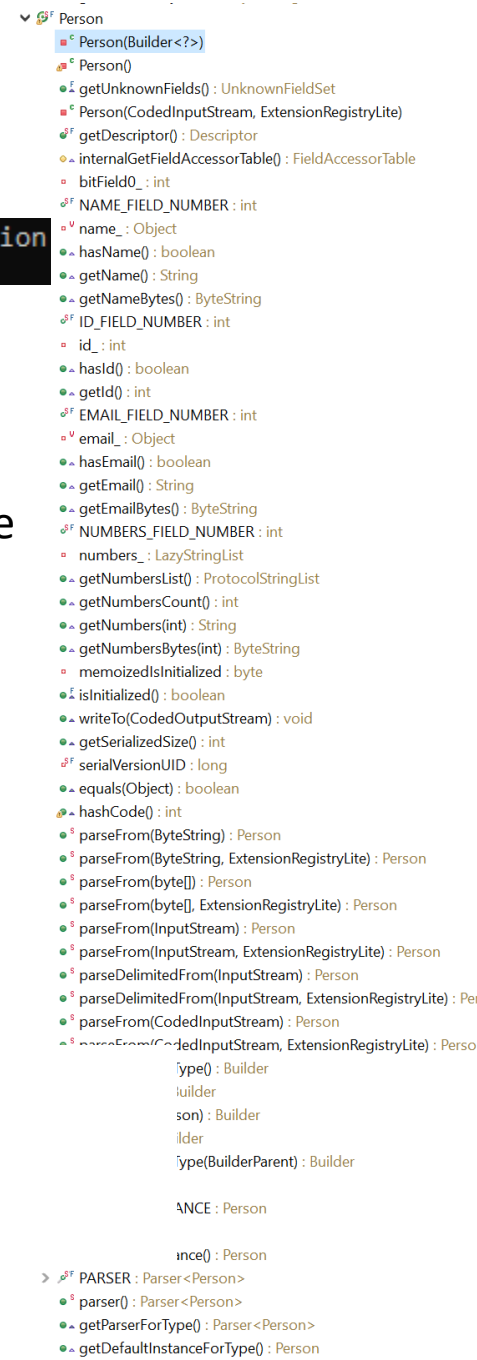
- Auto-generated code utility methods (chain-methods, etc.) help you managing serialization & deserialization

- Generated code and protobuf-jars will be part of your application as a dependency

- See (click on) migration guide proto2 to proto3 messages

protobuf-migration-from2-to-3

```
./
+- pom.xml
+- src/
   +- main/
      +- proto/
         +- message.proto
   +- test/
      +- proto/
         +- test_message.proto
```



Parsing and Serialization

Each protocol buffer class has methods for writing and reading messages of your chosen type using the protocol buffer binary format.

- byte[] **toByteArray**();: serializes the message and returns a byte array containing its raw bytes.
- static Person **parseFrom(byte**[] data);: parses a message from the given byte array.
- void **writeTo**(OutputStream output);: serializes the message and writes it to an OutputStream.
- static Person **parseFrom**(InputStream input);: reads and parses a message from an InputStream.

Protocol buffer classes are basically data holders (like structs in C) that don't provide additional functionality; they don't make good first class citizens in an object model. If you want to add r**icher behavior (needed for InterTest)** to a generated class, the best way to do this is to wrap the generated protocol buffer class in an application-specific class. **You should never add behavior to the generated classes (immutable) by inheriting from them**.

Extending a Protocol Buffer

Sooner or later after you want to "improve" the protocol buffer's definition. If you want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible – and you almost certainly do want this – then there are some rules you need to follow. In the new version of the protocol buffer:

- you must not change the **tag numbers of any existing fields**.
- you must not add or delete any required fields (proto2 messages).
- you may delete optional or repeated fields.
- you may add new optional or repeated fields but you must use fresh tag numbers (never used in this protocol buffer, not even by deleted fields).

Advanced Usage

Services - Protocol Buffers are similar to the Apache Thrift, Ion, and Microsoft Bond protocols, offers gPRC

Reflection - One key feature provided by protocol message classes is reflection. You can iterate over the fields of a message and manipulate their values without writing your code against any specific message type. One very useful way to use reflection is for converting protocol messages to and from other encodings, such as XML or JSON. A more advanced use of reflection might be to find differences between two messages of the same type, or to develop a sort of "regular expressions for protocol messages" in which you can write expressions that match certain message contents. Reflection is provided as part of the Message and Message.Builder interfaces.

Best Practices

- *Use PascalCase (with an initial capital) for message names – for example, MyServerRequest*
- *Use lower_snake_case for field names (including one of field and extension names) – e.g. , **title_name**. Respective JAVA or other language specific conventions will be used during the code generation, e.g. java.lang.String **getTitleName**();*
- *Do not re-use TAG(serial) numbers even deleted one, better make them **reserved** by tag_name or field_name, e.g.* reserved **2, 15, 9 to 11;** reserved "foo", "bar";
- *First TAG numbers 1-15, encoded with 1 byte, then 2 bytes, so consider keeping fields optimized*
- *Combining messages (e.g. message, enum, map, and service) in single large .proto file, may lead to dependency bloat*
- *Add comments via //, or /*…*/ syntax*
- *Protoc compiler generates code for your chosen language ([.h, .cc], [.java], [.ht], [.pb.go], [.rb], [.cs], …)*
- *Scalar types: double, float, int32|64, uint32|64, sint32|64, fixed32|64, sfixed32|64, bool, string, bytes[ByteString -* Provides conversions to and from byte[], String, ByteBuffer, InputStream, OutputStream. Also provides a conversion to CodedInputStream*] .*
- *Non-scalar types: Map, List (with repeated keyword), enum, nested classes, and compound types (**oneOf, Any**)*
- *Removing enum values is a breaking change for persisted protos, better mark it:* PHONE_TYPE_WORK = 3 [deprecated=true];
- *Extensions: reuse definitions via importing e.g.* import "school.proto";
- *Precisely, Concisely Document Most Fields and Messages*
- *More about updating message type , unknown fields*

Encoding	Sample types	Length
varint	int32, uint32, int64	Variable length
fixed	fixed32, float, double	Fixed 32-bit or 64-bit length
byte sequence	string, bytes	Sequence length

Data Type	Default value
Int32 / Int64	0
Float/double	0.0
String	Empty string
Boolean	False
Enum	First Enum item, that is the one with "index=0"
Repeated type	Empty list
Map	Empty Map
Nested Class	null

FlatBuffers

Zero-copy serialization format – designed for shorter serialization time and use less memory than protobuf.

FlatBuffers (download compiler here) is an efficient cross platform serialization library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift. A FlatBuffer is a binary buffer containing nested objects (structs, tables, vectors,..) organized using offsets so that the data can be traversed in-place just like any pointer-based data structure.

- Define a schema as .fbs files, and generate a code for favorite language
- We use generated final class (with factory methods and getters) to do serialization.
- Memory efficiency and speed
- Much efficient for large messages
- Flexible, tiny code footprint
- Strongly typed
- Cross platform code with no dependencies
- Access to serialized data without parsing/unpacking

- Usage is less practical than protobuf
- Documentation is less explanative, oldy
- More used in C/C++ e.g. gRPC

Built-in scalar types are:

- 8 bit: `byte ubyte bool`
- 16 bit: `short ushort`
- 32 bit: `int uint float`
- 64 bit: `long ulong double`

Main types

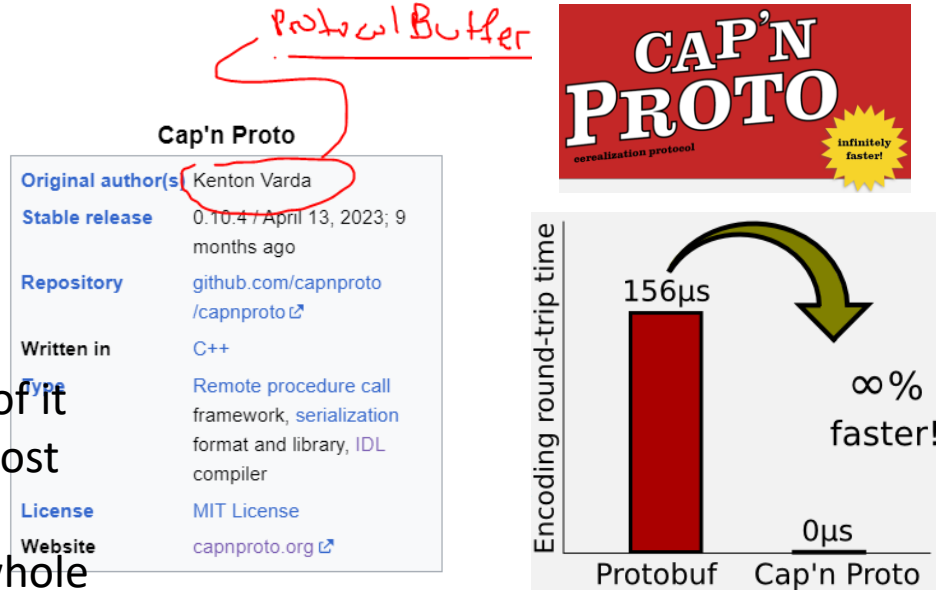
- A spectrum of basic types
- Structures with mandatory fields.
- Extensible tables with optional fields.

- See data types
- FlexBuffers - The schema-less version of FlatBuffers have their own encoding.

FlatBuffers

Original author(s)	Wouter van Oortmerssen
Developer(s)	Derek Bailey
Initial release	June 17, 2014; 9 years ago[1]
Stable release	23.3.3 / March 3, 2023; 10 months ago[2]
Repository	github.com/google/flatbuffers
Written in	C++
Operating system	Android, Microsoft Windows, Mac OS X, Linux
Type	serialization format and library, IDL compiler
License	Apache License 2.0
Website	flatbuffers.dev

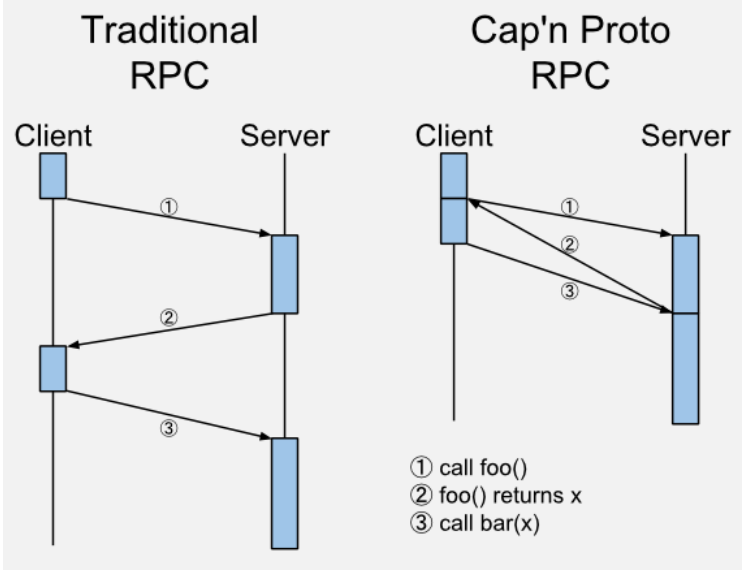
Cap'n Proto



[Cap'n Proto](#) is an insanely fast [data interchange format](#) and capability-based RPC system.

- Incremental reads - easy to start processing message before receiving all of it since outer objects appear entirely before inner objects (as opposed to most encodings, where outer objects encompass inner objects).
- **Random access** - can read just one field of a message no need to parse whole
- mmap - Read a large Cap'n Proto file by memory-mapping it.
- Inter-language communication: Two languages can easily operate on the same in-memory data structure. E.g. calling C++ code from, Java or Python.
- Inter-process communication - processes running on the same machine can share a Cap'n Proto message via shared memory. No need to pipe data
- Tiny generated code - Protobuf generates a lot whereas Cap'n code smaller
- **Tiny runtime library:** Runtime library is much smaller die to Cap'n format.

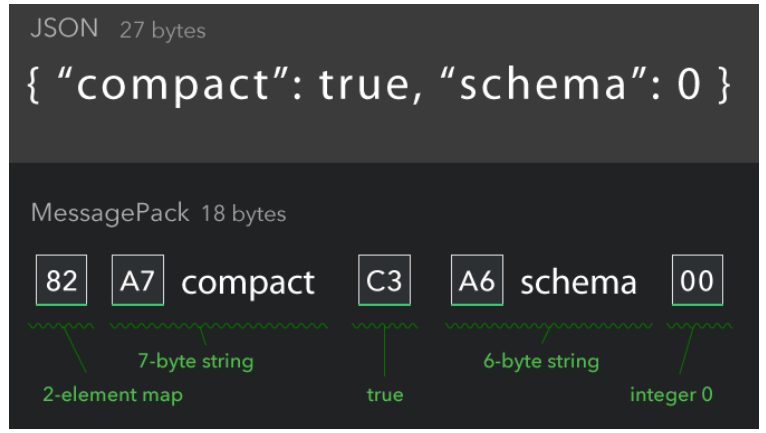
Time-traveling RPC: Cap'n Proto features an RPC system that implements [time travel](#) ([Promise Pipelining](#)) such that call results are returned to the client before the request even arrives at the server!



Symantec removed the installation win32 ;)

Message Pack

Zero-copy serialization format



MessagePack is a computer data interchange format. It is a binary form for representing simple data structures like arrays and associative arrays.

- MessagePack aims to be as compact and simple as possible.
- The official implementation is available for:
  C, C++, C#, D, Erlang, Go, Haskell, Java, JavaScript (NodeJS), Lua, OCaml, Perl, PHP, Python, Ruby, Rust, Scala, Smalltalk, and Swift
- Data types: nil, bool, int, float, str, bin, array, map, ext, timestamp

Limitations

- MessagePack is more compact than JSON, but imposes limitations on array and integer sizes. On the other hand, it allows binary data and non-UTF-8 encoded strings.
- Schema less format is lack of flexibility, less practical
- Compared to BSON, MessagePack is more space-efficient. BSON is designed for fast in-memory manipulation, whereas MessagePack is designed for efficient transmission over the wire.
- The Protocol Buffers format provides a significantly more compact transmission format than MessagePack because it doesn't transmit field names.

SBE

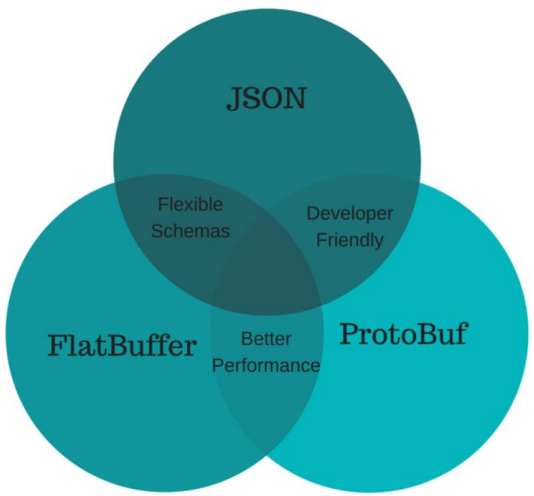
Simple Binary Encoding (SBE)

SBE is a binary representation for encoding/decoding messages to support **low-latency streaming**.
Ref-implementation ff FIX-SBE . SBE is an OSI layer 6 presentation for encoding and decoding binary application messages for low-latency financial applications.

The Message Structure
In order to preserve streaming semantics, **a message must be capable of being read or written sequentially, with no backtrack**. This eliminates extra operations — like dereferencing, handling location pointers, managing additional states, etc. – and utilizes hardware support better to keep maximum performance and efficiency.

• Header: It contains mandatory fields like the version of the message. It can also contain more fields when necessary.
• Root Fields: Static fields of the message. Their block size is predefined and cannot be changed They can also be defined as optional.
• Repeating Groups: These represent collection-type presentations. Groups can contain fields and also inner groups to be able to represent more complex structures.
• Variable Data Fields: These are fields for which we can't determine their sizes ahead. String and Blob data types are two examples. They'll be at the end of the message.

Feature	Protobuf	JSON	XML
Language independent	Yes	Yes	Yes
Serialized data size	Least of three	Less than XML	Highest among the three
Human Readable	No, as it uses separate encoding schema	Yes, as it uses text based format	Yes, as it uses text based format
Serialization speed	Fastest among the three	Faster than XML	Slowest among the three
Data type support	Richer than other two. Supports complex data types like Any, one of etc.	Supports basic data types	Supports basic data types
Support for evolving schema	Yes	No	No



- **Use Protobufs** if we want to be more efficient and the message is not that big (1 MB or less).
- **Use FlatBuffers** if we want to be more efficient with larger messages. FlatBuffers is the better choice if you're looking to create read-only query messages - this feature also saves on time and memory.

Tiny generated code: Protobuf generates dedicated parsing and serialization code for every message type, and this code tends to be enormous. Cap'n Proto generated code is smaller by an order of magnitude or more. In fact, usually it's no more than some inline accessor methods!

Feature	Protobuf	Cap'n Proto	SBE	FlatBuffers
Schema evolution	yes	yes	caveats	yes
Zero-copy	no	yes	yes	yes
Random-access reads	no	yes	no	yes
Safe against malicious input	yes	yes	yes	opt-in upfront
Reflection / generic algorithms	yes	yes	yes	yes
Initialization order	any	any	preorder	bottom-up
Unknown field retention	removed in proto3	yes	no	no

MessagePack vs Protobuf: What are the differences?

Both MessagePack and Protobuf are popular data interchange formats that offer compact and efficient serialization and deserialization of structured data. Let's explore the key differences between MessagePack and Protobuf.

1. **Data Representation**: MessagePack uses a binary format for data representation, whereas Protobuf uses a combination of binary and textual formats.
2. **Schema Definition**: MessagePack does not require a schema definition for serialization and deserialization, making it more flexible for loosely structured data. On the other hand, Protobuf requires a strict schema definition using a .proto file, enforcing a structured and strongly typed data model.
3. **Compatibility**: MessagePack supports backward and forward compatibility out of the box, allowing for seamless communication between different versions or implementations of the data format. In contrast, Protobuf requires careful management of schema evolution and versioning to ensure compatibility.
4. **Language Support**: MessagePack has extensive language support, with libraries available for many programming languages. Protobuf also has broad language support, but it is primarily focused on the languages supported by Google, such as Java, C++, and Python.
5. **Serialization Efficiency**: MessagePack provides a highly efficient binary serialization format, resulting in smaller message sizes. Protobuf also offers efficient serialization but may have slightly larger message sizes due to the inclusion of field tags and length prefixes.
6. **Rich Feature Set**: Protobuf offers a richer feature set compared to MessagePack, with support for features like default values, nested messages, and enums. MessagePack, on the other hand, provides a more minimalistic approach, focusing on simplicity and performance.

Use Case: Exchange InterTest messages via network, have state and in some cases contain actions(service methods), message sizes are max 4-7KB.
Conclusion: For smaller messages (especially < 1-2 MB) Protobuf has the best performance. Actions can be done via wrapper or gPRC solution in Protobuf. Also it has big community feedbacks on different use-cases. It also offers more advanced features like (rich types, extensions, services, etc.)
Cap'n Proto is other option (only wins over Protobuf for big messages), it is new and claims best, less community, main users are focusing on C/C++