SIKE: An Analysis of Serializing Interfaces in Kafka Experiments

Sahil Gandhi, Kaushik Mahorker, Wenlong Xiong
University of California, Los Angeles
{sahilmgandhi,kaushikm,wenlongx}@ucla.edu

Apache Kafka is a distributed streaming platform that is widely adopted today for aggregating and analyzing huge flows of real-time data. One of the key characteristics of Kafka is the ability to be protocol agnostic for serializing and deserializing data as one can override the SerDe interface, but the default that is suggested to the user is Avro. We have created the SIKE engine to profile various binary protocols for their serializing/deserializing times and also compare their relative performance in a Kafka consumer/producer cluster. We evaluate on a variety of metrics such as the producer send rates, consumer lags, broker request latency, and more. We have found that perhaps a new serializer, Cap'n Proto, can take the reigns for being default the serializer of choice for Kafka experiments.

1 Introduction

Being data driven is the new norm in any corporate culture today, whether it be in a technology firm, a finance firm, or even in the hospitality industry. A rather critical driving factor behind this is the sheer amount of data that can be gathered from users from their computers, smartphones, and IoT devices. These data schemas evolve over time and thus any service that consumes the data and processes it must be smart enough to work with this change and still produce consistent results. In addition, the data needs to be processed in an efficient manner conserving latency and size to meet the needs of the rising volume of data.

In light of this, we leverage the extensible SerDe interface of Apache Kafka to implement custom serializers and deserializers using four protocols. We explore the performance of these protocols with respect to metrics such as latency, size, byte-rate and records-lag to help developers make more informed design decisions for their data driven applications. Our results show that while Avro is extremely performant, Thrift and Cap'n Proto offer better serializing and deserializing times and virtually no message lag on the consumer end.

The rest of the paper is organized as follows. We provide a brief overview of the relevant components of Kafka in Section 2 and related work in serialization profiling in Section 3. In Section 4, we look at differences between the Binary formats we chose to evaluate. We describe the system and design of the profiler in Section 5 and provide an evaluation of the different protocols as well as insights in Section 6. In Section 7 and 8, we discuss potential future work and conclude with our final remarks.

2 Background and Motivation

Apache Kafka is a distributed streaming platform that lets multiple producer and consumer clients durably read and write streams of data like a messaging system [12]. Since its introduction by LinkedIn in 2011, it has become widely used in a variety of data-driven applications, including as a drop-in replacement for traditional message brokers, log aggregation, metrics reporting, stream processing, event sourcing, as an external commit-log, etc. Most common use cases of Kafka involve sending large volumes of structured data that follow slowly changing schemas. Driven by this usage pattern, many companies opt to send data in a compact binary format, serializing and deserializing them at the producer and consumer clients.

The use of binary protocols offer many benefits over sending unstructured data. Serializing data to a binary format allows for compact storage and lower memory usage, reduced network overhead, language-agnostic data transfer (if the clients also implement the serialization/deserialization interface), as well as the enforcement of message schemas. While officially, Apache Kafka does not enforce any requirements on the data transmitted (opting instead to represent messages as pure byte arrays), there are many third-party libraries that provide serializers/deserializers for various binary formats.

While the use of binary protocols offer many benefits over unstructured data, they also incur overheads from the serialization / deserialization process. In addition, not all compact binary formats are created alike. Currently, the only first-party Kafka client serialization/deserialization implementation is

for Apache Avro, which is provided by Confluent, a company founded by the original authors of Apache Kafka. Since Apache Avro was first developed in 2009, other protocols have been developed that offer increased performance in either speed, storage size, or flexibility. While Avro has been a default choice for good overall performance, switching to another format may offer different trade-offs that are more beneficial for certain data pipelines.

However, as far as we know, there have been no empirical comparisons of existing binary formats and their performance in the context of Kafka. Our work aims to evaluate the performance of some of these binary protocols for a few different typical Kafka workloads, and compare them on key metrics such as serialization/deserialization speed, storage size, and ease of use.

3 Related Works

While there have been no empirical comparisons of existing binary formats in the context of Kafka, there have been several comparisons of existing binary formats in blog posts and Github gists.

One such comprehensive blog post is from Uber [11]. They compare a wide variety of serialization technologies and their performance together with different compression libraries. While the article is comprehensive, it was written in 2016, and they are exploring the technologies of 2014 and earlier since they are looking retroactively at what Uber "may have done" in the past. Uber's analysis is based on outdated assumptions (Protobuf v3 has come out since, Avro has been incremented by five versions, and newer technologies like Cap'n Proto exist today), and should only serve as a starting point into serializer analysis. We wish to explore more modern serializers, as well as evaluate their performance in the usage context of Kafka in our paper.

Similar to the Uber blog post is the jvm-serializers repository on Github [7], which contains a comparison of more than ten different serializers, and presents the serialization/deserialization times on the wiki page. While the repository has been updated with modern versions of the serializers, the wiki image has not been updated since 2016 and are no longer up to date. Furthermore, they only evaluate based on a single testing schema, which contains both simple and complex (lists/nested messages) data types. We aim to provide a more up to date profiling of the serializers, and also evaluate their performance on different schema types to see if certain serializers are more performant based on the schema complexity.

Finally, the performance of different binary serializers is accentuated in a Google paper where they profiled the live traffic and performance of ware-house servers [10]. Here they observe that the overhead of serializing and deserializing data to a binary format (Protobuf in their case) takes up about 10% of the CPU cycles of an average warehouse machine. That is a significant portion of compute resources, and equates to millions of dollars worth of CPU time across all the data centers. If a different serialization format is ostensibly better for certain types of data, or better in general compared to other binary protocols, the performance implications can save a company much money. Our work aims to reveal any such underlying advantage.

4 Binary Formats

In our experiments, we evaluate three binary formats in addition to the existing Avro implementation.

4.1 Avro and Schema Registry

Apache Avro is currently the default choice of binary format for Kafka messages [1]. It was originally developed as part of Apache Hadoop, for compact binary storage, but eventually became a top-level project in its own right.

Avro requires users to define a schema (either using JSON or an IDL), which describe the data fields and their types. Avro stores its schema with the data it encodes, allowing clients to easily read encoded data, without the need for statically generated accessors. Because the schema is stored with the encoded data, type information and other metadata does not need to be encoded into the binary representation. Avro also allows for schema evolution through resolution rules, allowing for readers and writers of Avro to use different versions of schemas. Because of this, Avro requires the schema to be present both at serialization and deserialization time. In the case that the reader and writer schemas are different, Avro requires both schemas to be present. These attributes make Avro a good overall choice for many data intensive applications.

Confluent has made some additions to Kafka especially for Avro. In particular, Confluent defines a Schema Registry, which is a reliable and distributed service that allows clients to store and retrieve Avro schemas through a REST interface. This provides a central location for cross-language clients to store schema metadata, and facilitate the addition of new clients to existing data pipelines.

4.2 Protobuf3

Protobuf3 (also called protocol buffers or Proto3) is a binary format developed at Google, and is in its third major iteration [5]. We chose Protobuf to evaluate due to its widespread usage and ability to support schema evolution, as well as its low uncompressed message size (smaller than Avro according to Uber [11]) and fast encode/decode speed.

Like Avro, Protobuf requires users to define a schema, that allows for both simple and complex types. However, Protobuf differs from Avro in that it requires developers to define the

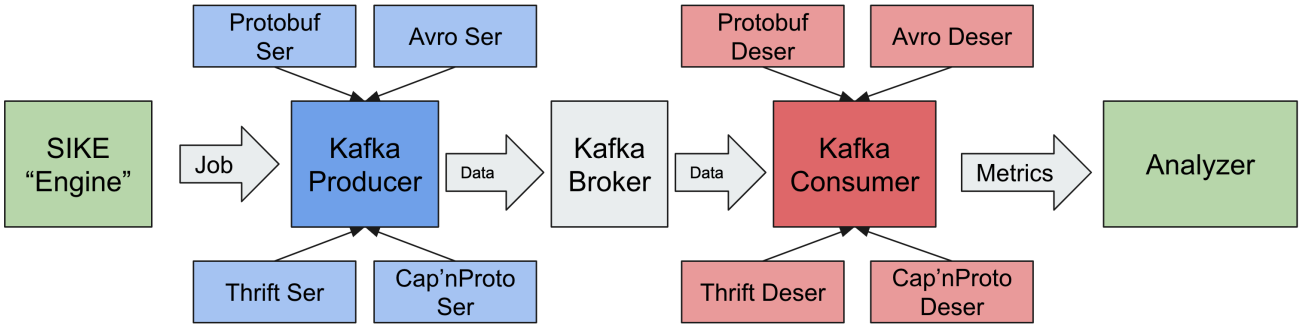

Figure 1: Overview of the entire system

schema in a definition file, and use a separate compiler to generate the language-specific classes and message headers for the data. In addition, Protobuf encodes some metadata into the binary format, including an ID and type for each data field. This means that Protobuf also allows for schema evolution through resolution rules; a newer version of a schema is compatible with an older one as long as the field IDs match. This is both an advantage and disadvantage: using a single compiler means that cross-language clients are generated from a single source, but means that static code gen is required; since Protobuf stores field metadata in the binary representation, it does not need to store the entire schema in the binary format, but requires clients to have generated accessors from the Protobuf compiler.

4.3 Cap'n Proto

Cap'n Proto is a binary format created by former Google engineers to address the serializing and deserializing constraints that most modern serializers have [3]. We chose to evaluate Cap'n Proto as it claims to be infinitely times faster than Protocol Buffers (or the equivalent) and thus can save much time in the serializing/deserializing steps.

Cap'n Proto directly interacts with objects stored in ByteBuffers, stepping over any serializing or parsing step directly. It however does require developers to define the schema in a definition file and uses a separate compiler to generate the language-specific classes and message headers for the data. The encoding of data formats and metadata is also sent in this binary format, which allows for schemas to be more flexible and evolve. Any new fields that are added are merely appended to the end of the byte buffers with the appropriate metadata. Furthermore, the generated classes allow languages to interact with the raw bytes in a type-safe manner. Since Kafka requires byte arrays (byte []) we will not get to exploit the "non-serialization" advantage of Cap'n Proto, and will have to explicitly convert ByteBuffers to byte arrays.

4.4 Thrift

Apache Thrift (originally developed at Facebook) is a binary format coupled with a RPC framework [2] We chose to evaluate Thrift due to it having similar if not better encode/decode speed to Protobuf [11] (possibly at the cost of message compaction), while having better RPC support.

Thrift is similar to Protobuf in terms of usage: clients define message fields in an IDL, and Thrift uses a single compiler to statically generate client code to serialize and deserialize messages to a binary format. The binary representation stores message field types and metadata, so Avro supports schema evolution. Thrift differs from Avro and Protobuf in that it supports multiple binary encoding formats optimized for different usages (space efficiency, fast encode/decode speed, etc). It also includes several different transport protocol implementations, and immediately supports RPC using the Thrift server capability.

5 System Overview

We leveraged Kafka's extensible SerDe interface to create a profiler for custom serializers and deserializers. All components are written in Java8 except for the analyzer/visualizer which is in Python3.

1. The SIKE execution engine that kicks off an experiment for a particular serializer. We have included support for Avro v1.9.1, Thrift v0.13.0, Protobuf v3.10.0, and Cap'nProto v0.1.4. The execution engine is also in charge of varying the iterations (ie. number of messages created) and picking what kind of message is sent. The message types are as follows:

   (a) Simple message - This message type consists of just primitive data types such as ints and strings.

   (b) Complex message - This message type consists of more complex data types such as arrays and maps.

(c) Nested message - This message type consists of using other nested messages.

2. A Kafka producer that creates the messages of the type defined by the engine, and using the custom serializer denoted by the engine as well. It creates a fresh object for each iteration, thus acting like a vanilla producer, rather than creating and caching an object once and sending it out each time. The producer's serializer also computes the serialized time for each message and writes it to a CSV file. Furthermore, it also captures producer-centric Kafka metrics and outputs them in JSON format to a file.

3. A Kafka consumer that consumes the messages of the type defined by the engine, and using the custom deserializer denoted by the engine as well. It computes the deserialization time for each message and writes it to a CSV file. Furthermore, it also captures consumer-centric Kafka metrics and outputs then in JSON format to a file. Both the producer and the consumer run in separate threads from the main engine.

4. A Python script that takes all of the generated CSV files and runs some analytics on the data. It also produces the graphs that are seen in the evaluation section below. We could have implemented this in Java as well, but Python provides some useful libraries for data processing and visualizations.

6 Evaluation

We conducted an experimental study, comparing two different performance classes. The first performance class is the raw serialization and deserialization times for each of the serialization technologies. We used three different types of messages for each tech, which are shown in Figure 2. We also varied the amount of messages created in each experiment, ranging from {1k, 3k, 6k, 10k, 25k, 50k, 75k, 100k}. We ensure to isolate the serialization and deserialization times by starting and stopping a timer in our custom SerDe implementations, thus removing any overhead from IPCs and only capturing the time for the serializing/deserializing operations.

The second performance class is comparing Kafka producer and consumer metrics with each of the serialization technologies. We once again use the three schema types shown in figure 2 and vary the amount of messages created, ranging from {1k, 3k, 6k, 10k, 25k, 50k, 75k, 100k}.

We ran our experiments on a single MacOS machine, with a dual core 2.4 GHz processor, 8 GB of RAM, and 256 GB SSD. The single machine hosted the zookeeper instance, the kafka broker, the producer and the consumer, with each running on a separate thread. the single machine sufficient for our study because we are solely interested in the relative performance between these different protocols. Controlling the environment we run these protocols on provides us with the same baseline that we expect to extend to a similar distributed setting. However we acknowledge that running on a single machine does not take into account certain factors that may affect the absolute metrics only present in a distributed setting.

```
message SimpleMessage {
  int64 timestamp = 1;
  string query = 2;
  int32 page_number = 3;
  int32 result_per_page = 4;
}

message ComplexMessage{
  int64 timestamp = 1;
  map<string, int32> storage = 2;
  repeated int32 arr = 3;
}

message NestedMessage{
  int64 timestamp = 1;
  int32 id = 2;
  SimpleMessage simpleMsg = 3;
}
```

Figure 2: The different message schemas used across the serializers. Protobuf syntax is shown above.

```
- Simple:
    - Avro: 100018 bytes
    - Protobuf: 100017 bytes
    - Capnproto: 100064
    - Thrift: 100019
- Complex:
    - Avro: 878 bytes
    - Protobuf: 1200 bytes
    - Capnproto: 2848 bytes
    - Thrift: 877 bytes
- Nested:
    - Avro: 100022 bytes
    - Protobuf: 100023 bytes
    - Capnproto: 100088 bytes
    - Thrift: 100031 bytes
```

Figure 3: Serialized message sizes for the various message types and serializers.

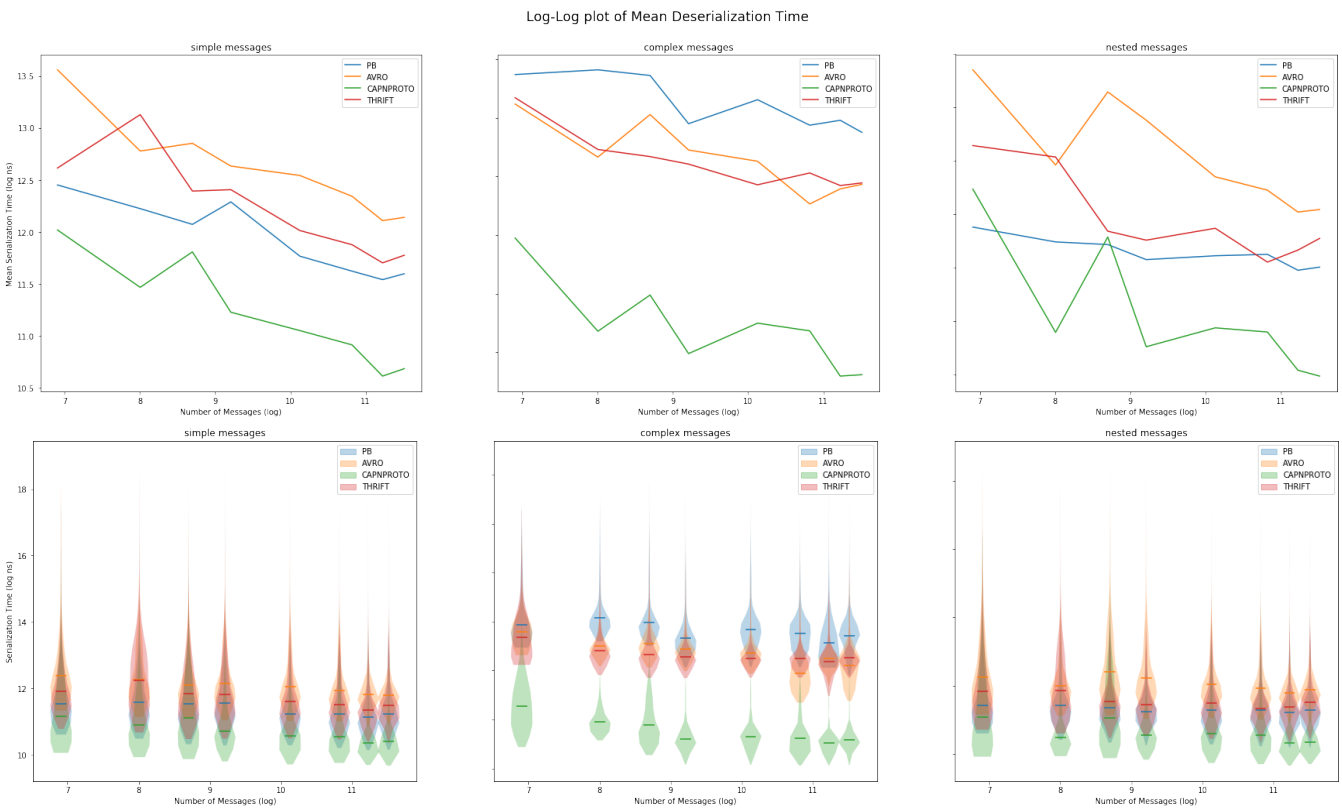

Figure 4: Comparison of serialization times for different binary formats.

6.1 Serializing Time

The first metric we measure is the serialization time across the protocols and across the different types of messages. Figure 4 plots the performance of these different protocols in relation to the amount of messages sent. We measure how well the different protocols handle messages of varying magnitudes to test their scalability.

One key observation is that as we increase the number of messages we see a larger separation in median serialization time. The additive nature of the delay in serialization time allows us to make greater distinctions about the different protocols asymptotically. In other words, larger samples in terms of the number of messages give us a better view of how serialization times are distributed.

From the figures we can ascertain a few empirical truths. Cap'n Proto is the clear leader and is able to serialize messages several times faster than that of the other protocols consistently across the different message types, by a large margin. Furthermore, Cap'n Proto scales well with the number of messages as it remains fairly consistent. However, more surprising is that Avro, the Confluent default schema format for Kafka, has average to low performance in comparison to other serializers across different message types. It has the worst average performance for simple and nested message types, and the second worst for complex messages (better than Protobuf). From the violin plots, we can see that there is a long tail of serialization times for each of the binary formats, but even then, the median time still implies that Avro has low performance across the board. Protobuf and Thrift have similar performance for all the different message types, with Thrift performing slightly better for complex messages. However, they both have much slower serialization speeds than Cap'n Proto.

6.2 Deserializing Time

The next metric we measure is the deserialization time across the protocols and across the different types of messages. Knowing the serializing time of these different protocols, one would expect to see something very similar in the deserialization times.

Figure 5 plots the performance of these different protocols in relation to the amount of messages sent. It shows that Cap'n Proto is able to deserialize messages at a faster rate than that of other protocols. This remains consistent across all message types. Particularly with the complex message type, no other protocols' distribution of deserialization times overlap with Cap'n Proto (per the violin plots). This illustrates the sheer maginitude that Cap'n Proto outperforms the other protocols. For deserialization, Confluent's default, Avro,

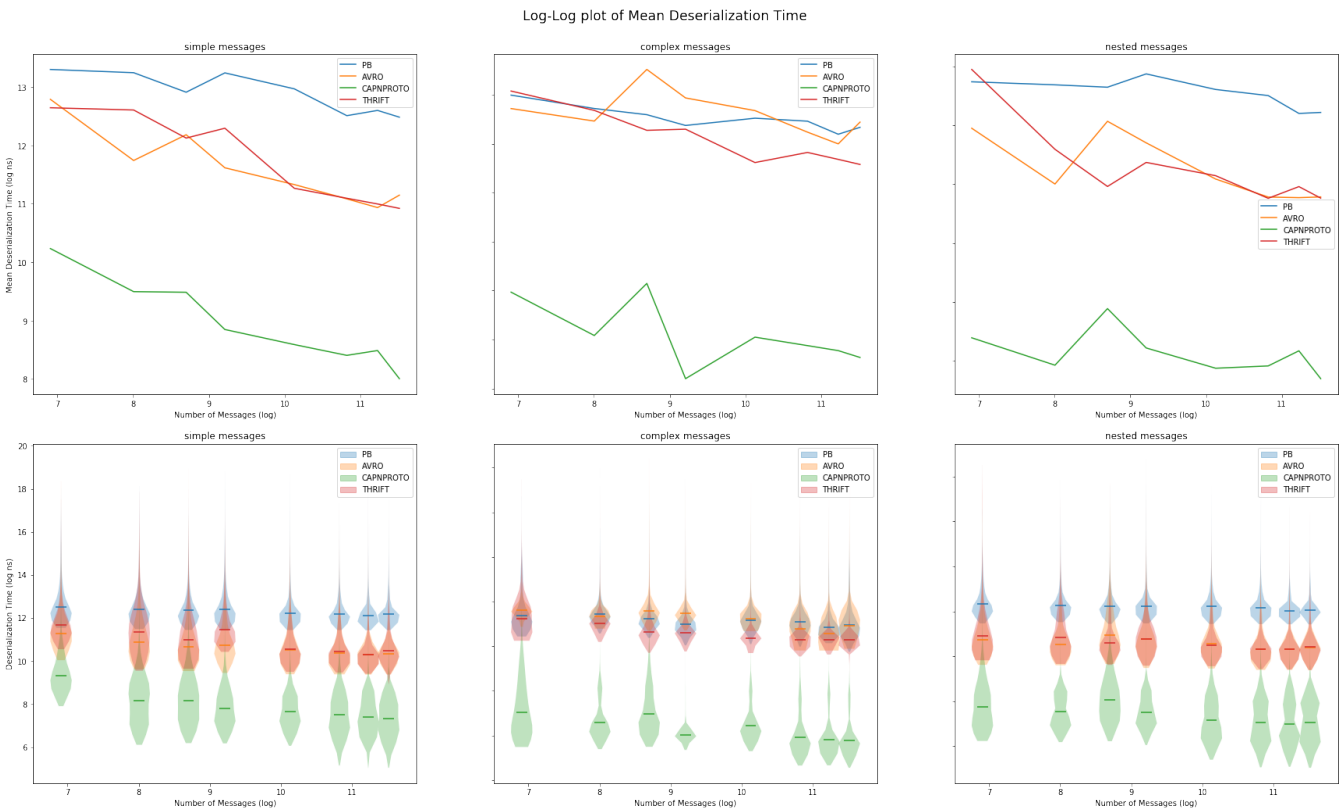

Figure 5: Comparison of deserialization times for different binary formats and message types.

performs very similarly to Thrift and is better than Protobuf for simple and nested messages, with Protobuf having similar performance for complex messages. Both the serialization times and deserialization times suggest that using Cap'n Proto is again the clear leader and a much better alternative than Avro for developers.

6.3 Kafka Metrics

Kafka provides a Metric class that tracks various metrics at the Producer and Consumer detailing the general performance of the Producer and Consumer. When these are measured across different protocols and message types, we discover some interesting insights.

6.3.1 Producer

On the producer side we request Kafka for the *request-latency-avg*, *buffer-available-bytes*, and *record-send-rate* metrics.

Request-latency-avg is a measurement of time-delay for producers to send serialized messages to the broker and receive an ACK. In turn, this serves as a proxy for CPU usage due to serialization overheads. As the producer is able to process messages faster, it is able to send messages to the broker faster as well. Therefore, a lower value for this metric is better. We can see from Figure 6 that most protocols have similar performance. Cap'n Proto, however, has increased latency values and this is due to Cap'n Proto having extremely fast serialization times. The larger request latency could be caused by an overwhelmed Kafka broker and thus Cap'n Proto's serialization speed is not capable of being fully leveraged by a single Kafka broker for a single producer/consumer.

Buffer-available-bytes is a measurement of how full the buffer is that the producer temporarily stores records in before sending it to Kafka. This serves as a measurement for what operations are slowing down the producer the most: if the buffer-available-bytes are low, that means producer-broker communication is the bottleneck; if buffer-available-bytes are high, that means operations like serialization are slowing down the system. We can see that for simple messages, the buffer is full virtually all the time, which means that network delays dominate; while for complex messages the buffer is not filled and the serialization process takes longer than writing and sending the records themselves. However, we can also see that of the protocols, Cap'n Proto results in a few sharp drops in buffer-available-bytes for complex messages, meaning that even for complex messages it has a low serialization overhead. Lastly, we can see that for nested messages, all protocols except for Avro (with schema registry) result in a full buffer, which implies that there is a noticeable overhead from using

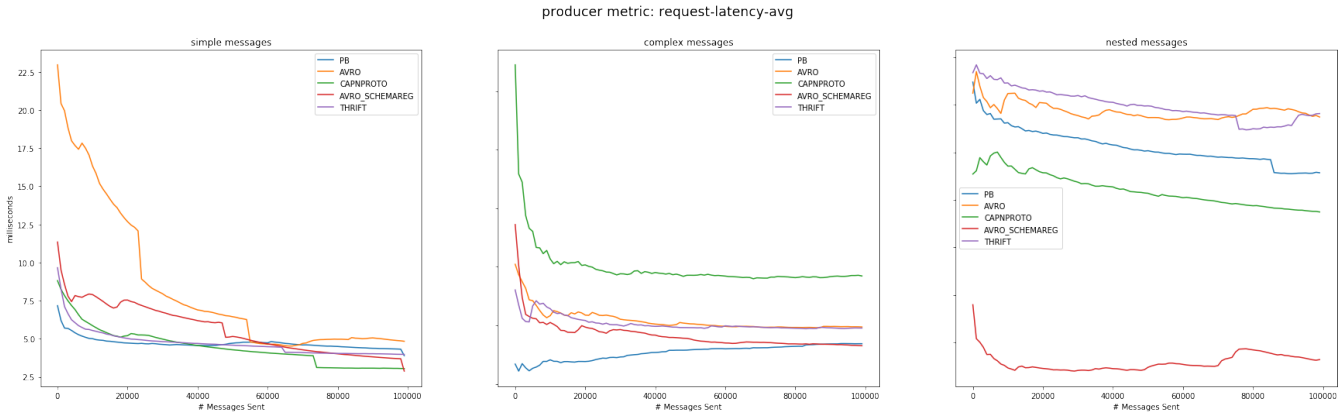

Figure 6: Average producer request latency, over an interval of 100k messages sent, for different binary formats and message types.

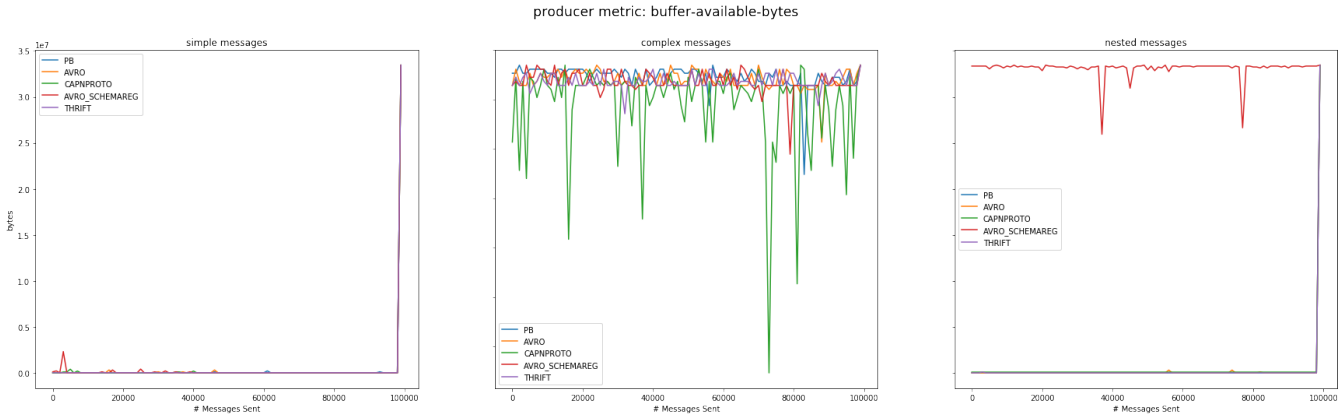

Figure 7: Available bytes in the producer buffer, over an interval of 100k messages sent, for different binary formats and message types.

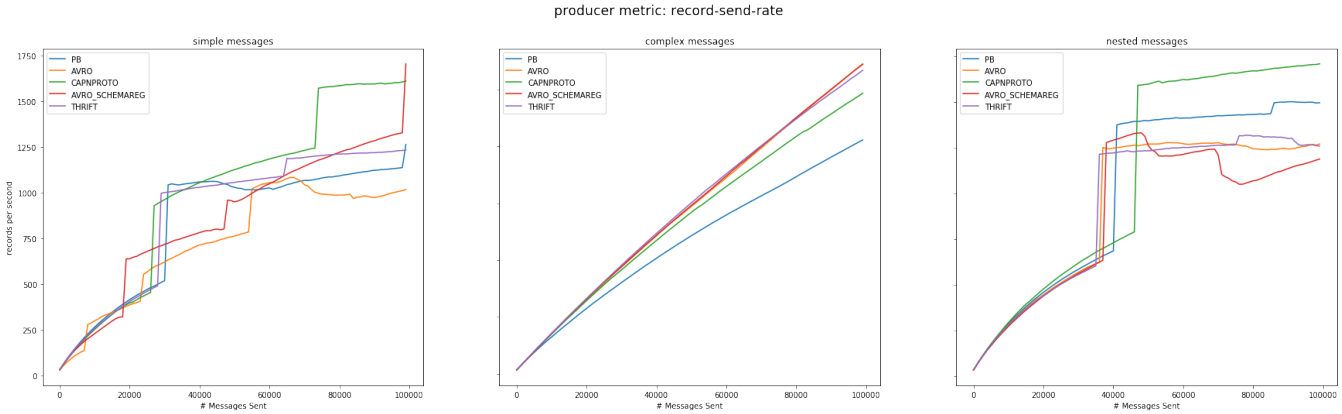

Figure 8: Average rate at which producer can send records, over an interval of 100k messages sent, for different binary formats and message types.

schema registry.

Record-send-rate measures the rate that records are pushed out from the producer in number of records per second and serves a proxy for message size and any overheads due to message transport from the protocols. This is a metric we want to be generally high as an increase in throughput means more

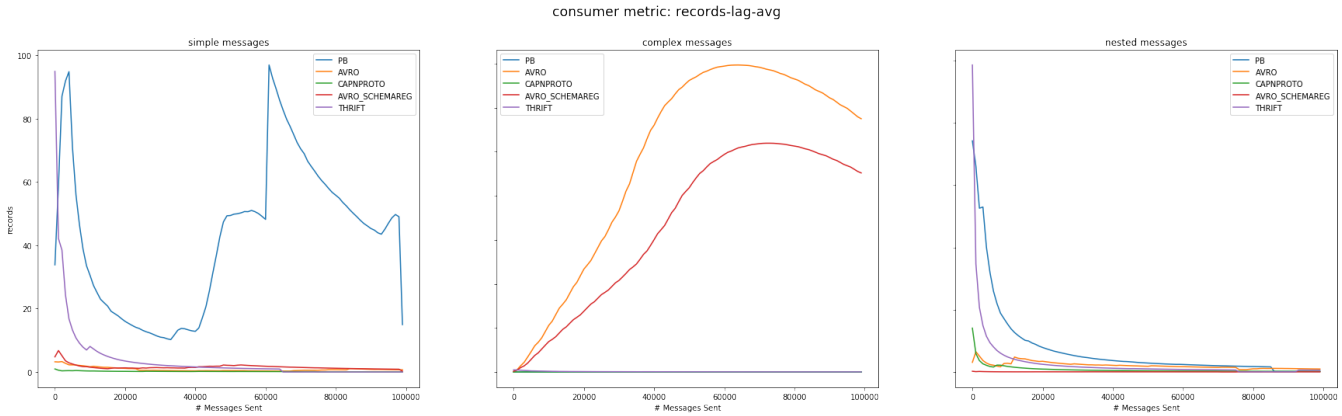

Figure 9: Average consumer lag, over an interval of 100k messages sent, for different binary formats and message types.

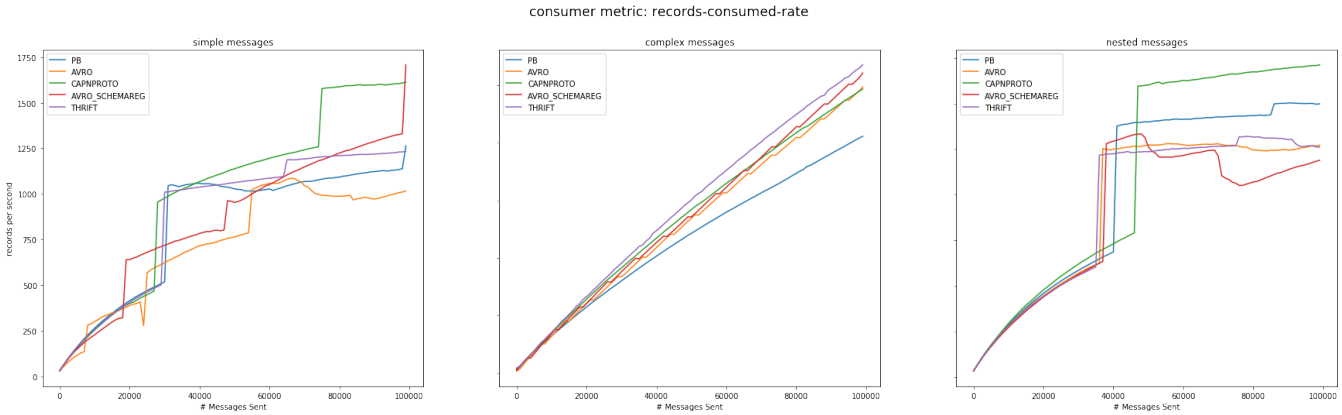

Figure 10: Average rate at which the consumer could consume records, over an interval of 100k messages sent, for different binary formats and message types.

messages are ready to be processed. From Figure 8 we deduce that Avro has the best message send rate for simple and complex messages and Thrift a close second. For nested messages Thrift leads consistently. Cap'n Proto's performance degrades for the nested message type for large numbers of messages, but this corresponds directly with the increased latency for the broker, so the overwhelmed broker may be held suspect here as well.

6.3.2 Consumer

On the consumer side we request Kafka for the *records-lag-avg* and *records-consumed-rate* metrics.

Records-lag-avg measures how much on average the consumer is behind the producer based on message offsets. For example if the message being consumed is at offset 50 and the message being produced is at offset 75, the *records-lag-avg* is 25. There are 25 messages waiting in the broker buffer yet to be consumed by the consumer. Keeping this metric low consistently indicates good consumer performance. This metric is also a proxy for relative comparison deserialization overhead as lower overhead means that the consumer can process records more quickly. Figure 9 shows that in general consumers have no lag, but there are a few cases when the consumer does fall behind. These cases specifically are Protobuf for simple and nested messaged and Avro for complex messages.

Records-consumed-rate, measures the rate at which records are pulled from the broker. Figure 10 is almost identical to the producer *records-sent-rate*. This makes sense because we will generally consume as many records as we can produce and no faster. This shows that the number of records that the consumer can consume will be directly affected by how well the producer can produce, thus placing a little more importance on serialization time than deserialization time.

6.3.3 Schema Registry Comparison

Finally, we also evaluated the overhead of using Confluent Schema Registry for dynamically keeping track of schema changes versus a native Avro serializer/deserializer that statically keeps track of schema changes (ie. when a new commit

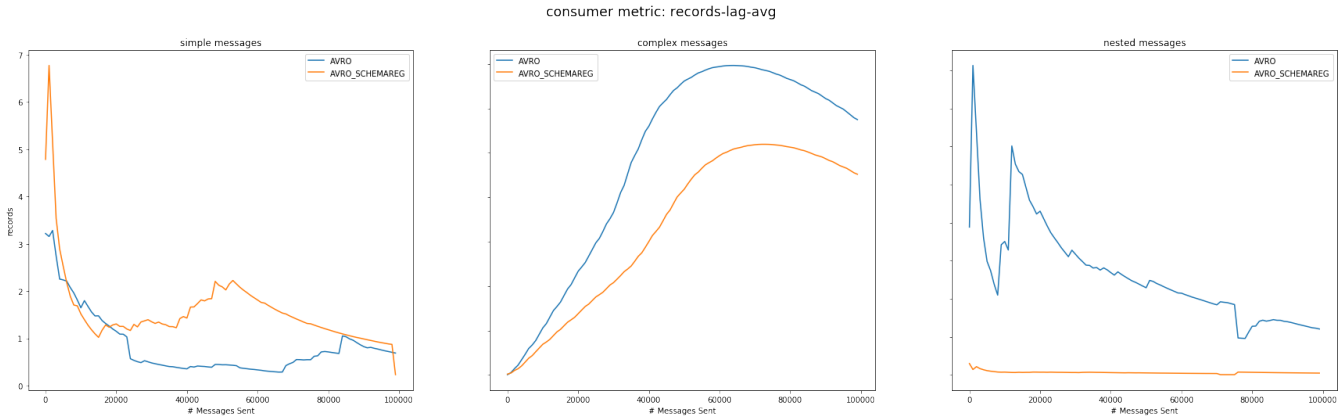

Figure 11: Difference between producer record send rate and consumer record receive rate, over an interval of 100k messages sent, for pure Avro vs SR and all three message types.

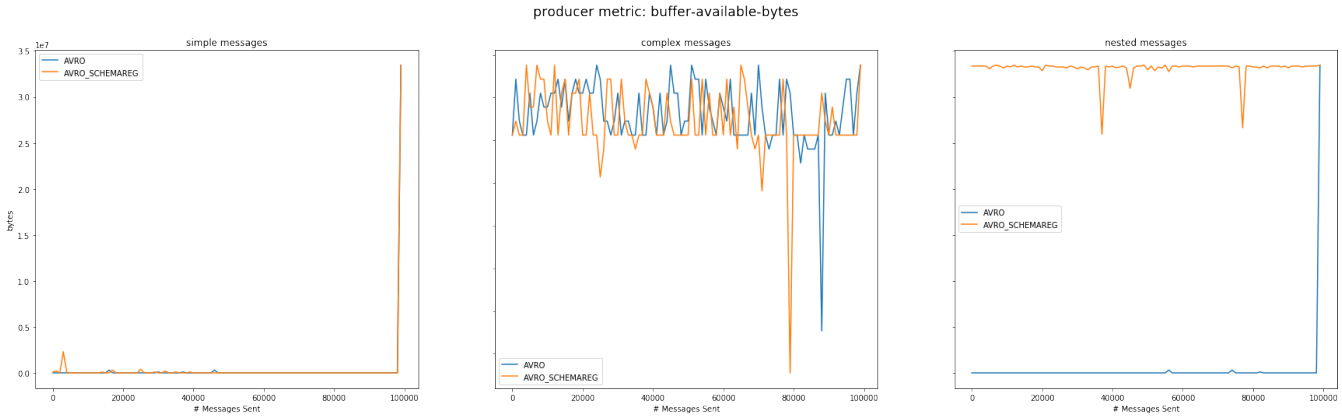

Figure 12: Available bytes in the producer buffer, over an interval of 100k messages sent, for pure Avro vs SR and all three and message types.

is pushed, it would check if the schema evolution is valid or not, but *not* do the check on every message). We choose to look at two metrics in particular here, the average record lag by the consumer 11 and available buffer bytes 12. We can see that for simple and complex messages, both have similar buffer sizes, but for nested messages, the Schema Registry version has a relatively empty buffer. This suggests, as aforementioned, that the serializing cost is greater than the network cost of transmitting the message. The lag paints a similar picture where we see that for both complex and nested messages, the native Avro producer is creating records at a faster rate than the consumer can receive, while the Schema Registry version which is unable to produce as fast, also has less of a lag on the consumer end. While these results do not quantify exactly how much overhead Schema Registry is adding, there is quite clearly a non-neglible overhead as the complexity of the message increases. Perhaps in a future experiment, we could repeat this experiment with several different schemas and versions of the schemas to force Schema Registry to perform more checks and thus give a more discernible difference (if any).

7 Future Work

While we have evaluated four different serializing technologies and measured how they perform in a Kafka cluster, there are still several other tasks that we could work on in the future.

The first is of course adding more serializers. While generally these four are the most popular ones that are used in industry today, there are still others that people may want to hook up with Kafka, and they may be curious on how they perform relative to other serializers. These include Google FlatBuffers [4], Kryo [8], Coffer [9], MessagePack [6] and BSON (Binary JSON). FlatBuffers are very similar in nature to Cap'n Proto, so we expect to see equivalent performance there, but are unsure how the others will perform. For example, Kryo is Java specific, but aims to be *even* faster than PB or Thrift at serializing and deserializing. We would also

compare how these serializers affect Kafka itself and when it would be more advantageous to use these versus the four that we profiled.

Furthermore, we evaluated the serializers with a simple Kafka producer/consumer cluster, but did not get a chance to do an in-depth for the Streams or Connect APIs or for Schema Registry (except for AVRO where Confluent has already implemented it). It is very possible that in these situations we may see different serializers come out triumphant due to data patterns in a stream, or overhead in schema-checking.

At the core, different serializers exist because they are able to work well for different use cases. Different message types may be more effective in certain use cases and thus different serializers need to be used. The flexibility of an extensible serialization/deserialization interface is what allowed us to conduct this study, but in a production environment the selection and configuration of serializers is an overhead that can be removed. Producers and Consumers can be extended to dynamically change their serialization/deserialization protocol based on the type of message and task at hand. Creating this abstraction would allow for faster development and avoid schema related overhead.

8 Conclusion

Serializers come in all shapes and sizes and Kafka can easily be integrated with any of them by implementing the Serializer and Deserializer interfaces. We evaluated the performances of Avro v1.9.1, Thrift v0.13.0, Protobuf v3.10.0, and Cap'nProto v0.1.4 in terms of serializing and deserializing time, as well as various metrics collected when run in a Kafka producer/consumer cluster. In terms of raw serializing and deserializing time, Cap'n Proto leads the pack by more than 25% in some message cases, while still keeping the forwards/backwards compatibility that is desired in binary protocols today. In terms of the Kafka metrics of interest such as the consumer records lag (ie. the number of messages the consumer trails relative to what the producer can create) we see that ProtoBuf creates the most consumer lag both simple and nested message formats, but Avro is the worst for the complex message format. Taking these added metrics into account, it still appears that Cap'n Proto is the protocol of choice, and perhaps a company like Confluent or avid open source developers may want to provide more first/third party support for the language! We have included all of our code at the following link: https://github.com/wenlongx/SIKE-CS239-Project.

References

[1] Apache avro.

[2] Apache thrift.

[3] Cap'nproto.

[4] Google flatbuffers overview.

[5] Google protocol buffers.

[6] Messagepack.

[7] Eishay. eishay/jvm-serializers, Jul 2019.

[8] EsotericSoftware. Esotericsoftware/kryo, Jul 2019.

[9] Gdotdesign. gdotdesign/coffer, Feb 2014.

[10] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[11] Kåre Kjelstrøm, Kåre Kjelstrøm, Kare, Kare, Uber, and Uber's Core Infrastructure. How uber engineering evaluated json encoding and compression algorithms to put the squeeze on trip data, Dec 2018.

[12] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. 2011.

A Graphs

We also evaluated several other metrics for the Kafka consumer and producer across all the binary formats and message types. Here are some graphs that we generated for those metrics.

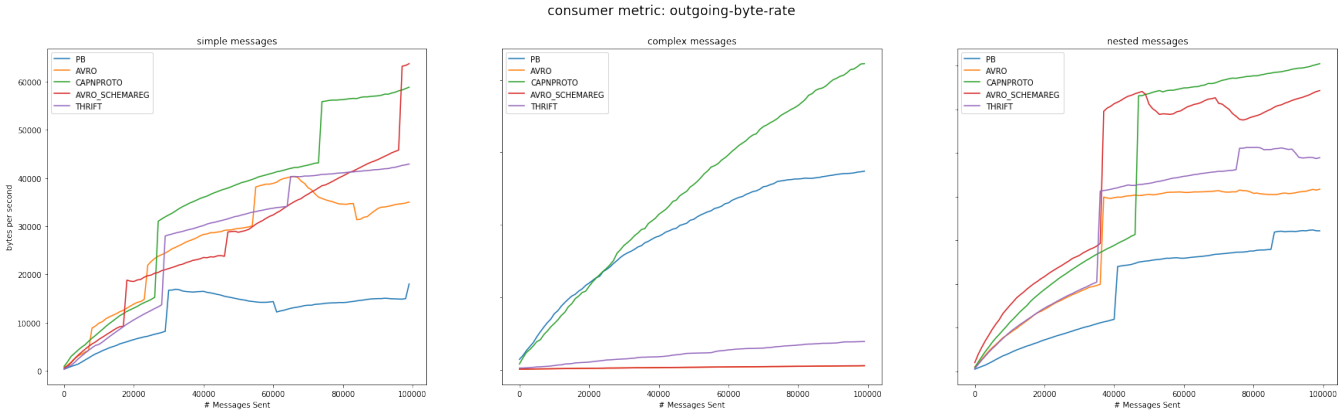

Figure 13: Average consumer outgoing byte rate, over an interval of 100k messages sent, for different binary formats and message types.

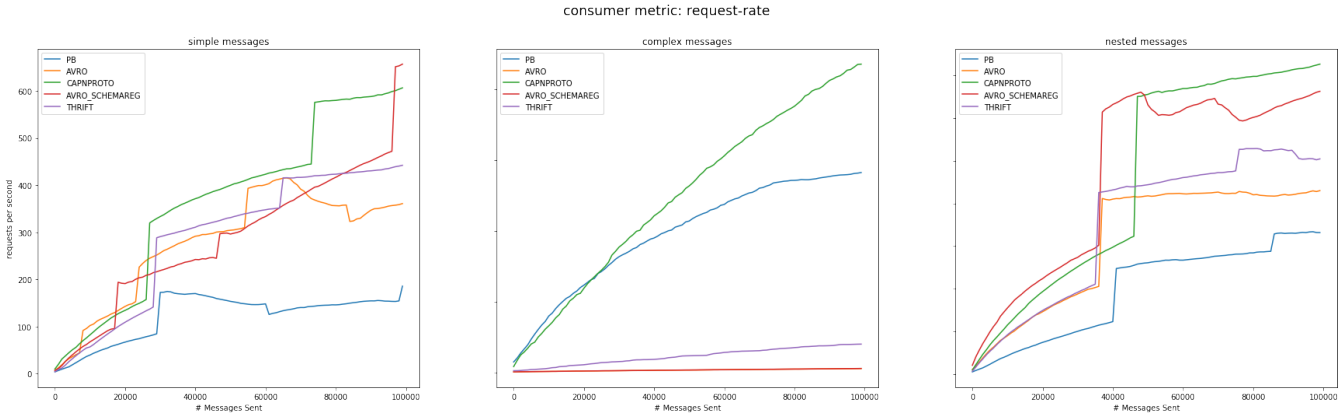

Figure 14: Average consumer request rate, over an interval of 100k messages sent, for different binary formats and message types.

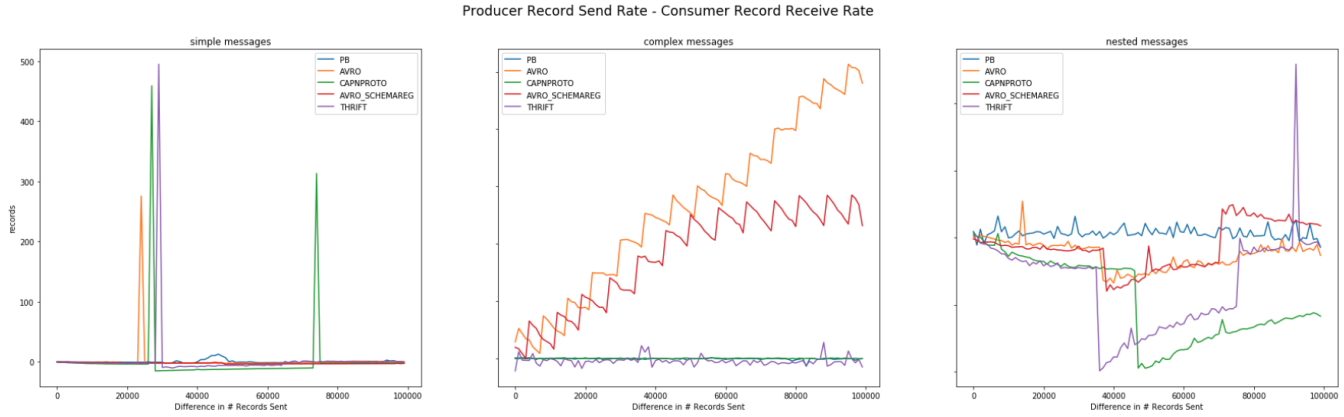

Figure 15: Difference between producer record send rate and consumer record receive rate, over an interval of 100k messages sent, for different binary formats and message types.

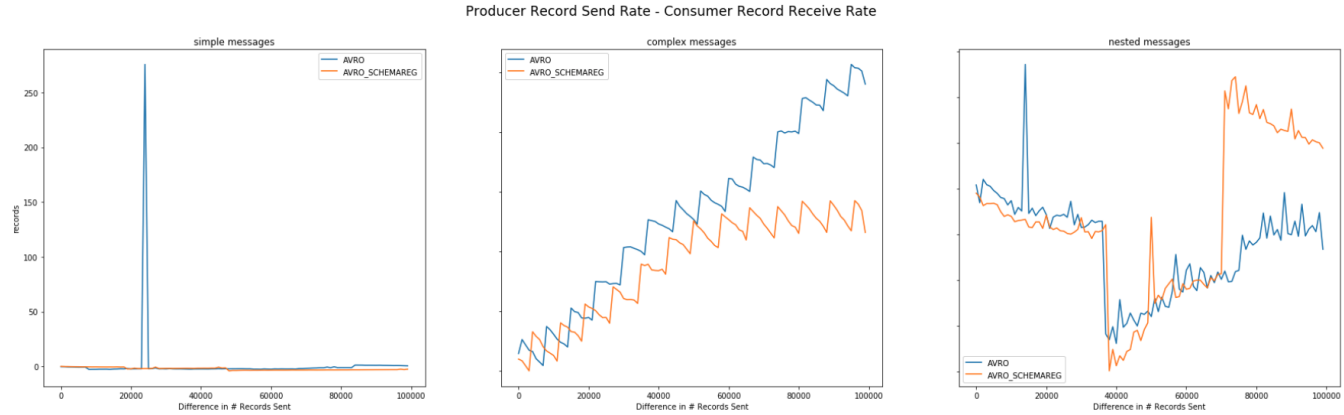

Figure 16: Difference between producer record send rate and consumer record receive rate, over an interval of 100k messages sent, for pure Avro vs SR and all three message types.

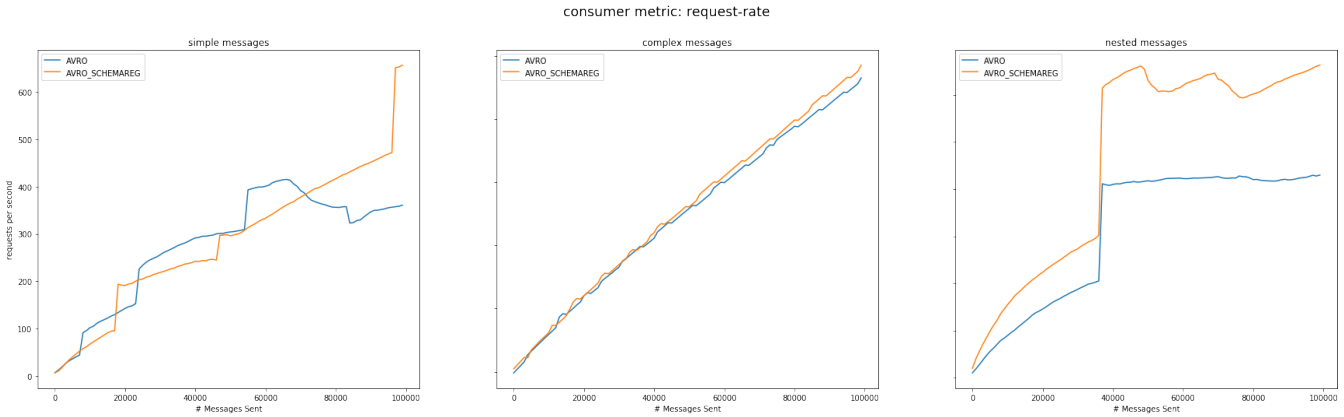

Figure 17: The consumer record request rate, over an interval of 100k messages sent, for pure Avro vs SR and all three and message types.

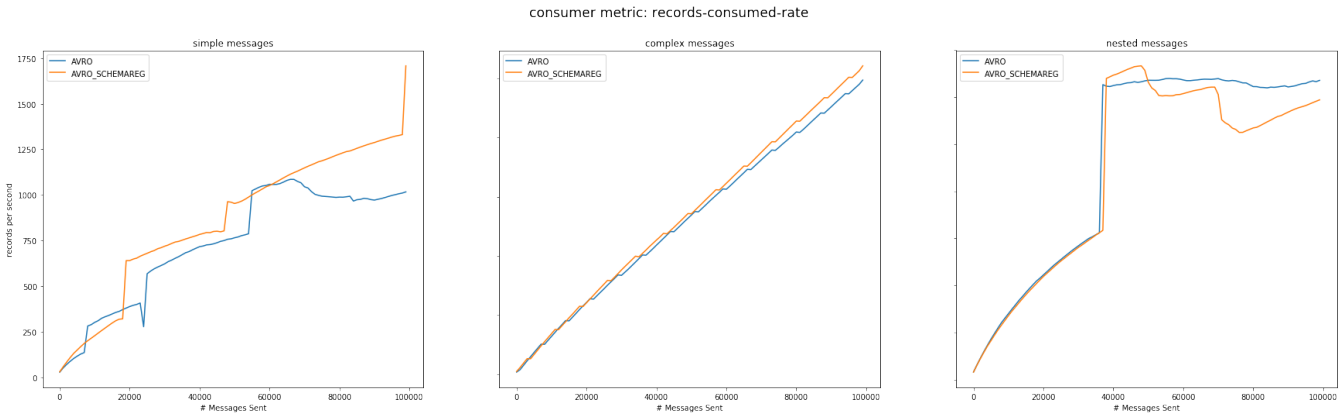

Figure 18: The consumption rate for records, over an interval of 100k messages sent, for pure Avro vs SR and all three and message types.

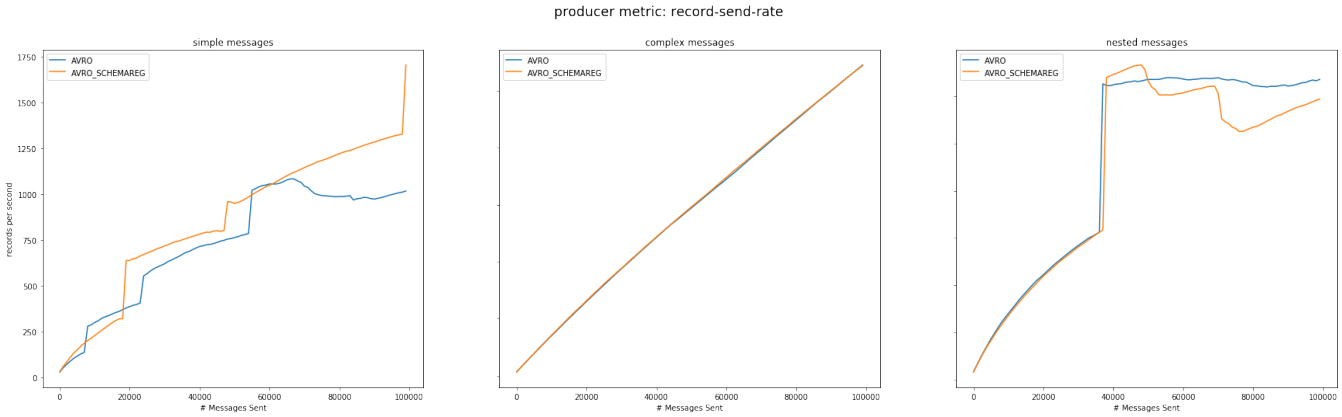

Figure 19: The producer's send rate, over an interval of 100k messages sent, for pure Avro vs SR and all three and message types.

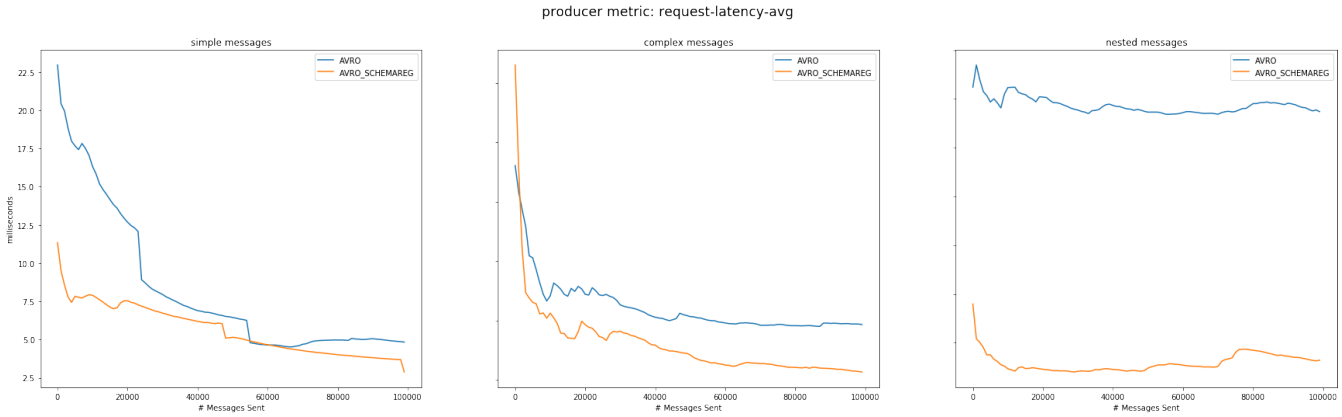

Figure 20: The average request latency, over an interval of 100k messages sent, for pure Avro vs SR and all three and message types.