ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 9, No. 1, 2024

BEYOND JSON: EVALUATING SERIALIZATION FORMATS FOR SPACE-EFFICIENT COMMUNICATION

Eduard Maltsev, Oleksandr Muliarevych

Lviv Polytechnic National University, 12, Bandera Str, Lviv, 79013, Ukraine.
Authors' e-mails: eduard.y.maltsev@lpnu.ua, oleksandr.v.muliarevych@lpnu.ua

https://doi.org/10.23939/acps2024.01.009

Submitted on 09.04.2024

***Abstract*: Distributed systems rely on efficient inter-service communication, heavily impacted by data transmission costs. This study investigates alternative serialization formats, like Avro and MessagePack, to reduce data size compared to the common JSON format. We utilize a custom model to comprehensively assess the space efficiency of serialization formats across various data types. Our findings demonstrate that adopting alternative formats achieves a median reduction in serialized data exceeding 30 %. Notably, Avro exhibits exceptional efficiency, leading to reductions exceeding 83 % in specific scenarios. These insights empower developers to select optimal formats, potentially leading to significant improvements in data transfer speed, reduced bandwidth consumption, and enhanced scalability for handling larger data volumes within distributed systems.**[1]

***Index Terms*: Data communication, Encoding, Information exchange, Protocols, Performance evaluation.**

I. INTRODUCTION

In distributed computing, the efficiency of inter-service communication stands as a cornerstone of system performance, scalability, and reliability. As distributed systems continue to underpin a growing array of critical applications from cloud computing and microservices to big data and IoT, the choice of serialization format for data interchange emerges as a pivotal consideration. Serialization formats, the mechanisms by which data structures are converted into a byte stream for storage or transmission, vary widely in their design, capabilities, and performance implications. The selection of an appropriate format is thus not simply an operational decision but a strategic one, bearing significant consequences for distributed architectures' overall efficiency and resilience. Despite the critical role of serialization in distributed systems, the field lacks a comprehensive, comparative analysis encompassing the wide spectrum of available formats, particularly considering recent technological advancements and the evolving demands of modern applications. Developers and system architects are often left to navigate this complex landscape with limited guidance, balancing trade-offs between speed, size, compatibility, and ease of use without clear, empirical benchmarks.

This study arises from the need to bridge this knowledge gap, offering a systematic evaluation of serialization formats within the context of inter-service communication in distributed systems. By scrutinizing a selection of widely adopted and emerging formats, this research aims to illuminate the characteristics of space efficiency, trade-offs, and practical considerations that inform the optimal choice of serialization technology. Specifically, the investigation targets formats categorized by binary or textual nature, schema requirements, and additional features such as zero-copy capabilities, addressing the nuanced requirements of diverse system architectures. The contributions of this study are manifold, offering actionable insights that promise to guide developers and architects in their selection of serialization formats, thereby enhancing the performance, scalability, and robustness of distributed systems.

Given the absence of exhaustive research addressing this topic, we aim to investigate whether alternative binary and textual serialization formats can reduce the serialized message size by at least 30% compared to JSON.

II. LITERATURE REVIEW AND PROBLEM STATEMENT

Let us examine recent studies to understand the current landscape of serialization formats and their implications for inter-service communication in distributed systems. Research in [1] contrasts JSON/XML with Protobuf for data serialization in web services, emphasizing efficiency, readability, and schema enforcement. JSON/XML is preferred in REST for its text-based, human-readable formats, enabling dynamic, schema-less data interchange. Another interesting study in [2] focuses on optimizing inter-service communication in a cloud-native microservice architecture. The study presents Protocol buffers as a language-neutral, platform-neutral, extensible mechanism for serializing structured data. They are known for their efficiency and performance benefits over traditional serialization formats like XML or JSON in certain use cases.

[1]This article uses the materials and results obtained by the authors during the research work "Intelligent design methods and tools for the modular autonomous cyber-physical systems," state registration number 0124U002340 dated 09.03.2024, which is carried out at the Department of Electronic Computing Machines of the Institute of Computer Technologies, Automation and Metrology of Lviv Polytechnic National University in 2024-2028.

The study [3] compares various serialization formats, focusing on vehicle-to-cloud communication. The paper evaluates Protobuf and Flatbuffers, two binary serialization formats. It mentions Cap’n Proto as an attractive zero-copy format, which performs similarly to Flatbuffers but with a slight speed advantage. Another alternative mentioned is MessagePack.

Source [4] explores binary versus textual serialization formats for inter-service communication within Java microservices under a K-Native, Kubernetes-managed environment. The study suggests that Protocol Buffers significantly improve response time and payload size performance.

Source [5] evaluates different serialization protocols for improving inter-service communication efficiency within dCache, a distributed storage system. It addresses the need to replace Java Object Serialization to enhance message-passing speed and reduce round-trip time.

Source [6] assesses diverse data serialization formats. The gap in this study's research is its focus on microcontrollers with certain constraints, which may not be generalizable to all IoT devices or distributed systems.

Source [7] stresses serialization formats' efficiency and performance in distributed systems, focusing on IoT sensor networks. Protocol Buffers or Apache Thrift were the most efficient means of encoding information based on the provided information.

Source [8] highlights several schema-driven and schema-less binary serialization specifications that are JSON-compatible, including but not limited to ASN.1, Apache Avro, Microsoft Bond, Cap’n Proto, FlatBuffers, and others.

Complementing these findings, [9] evaluates the performance impact of different communication protocols (REST, gRPC, and Thrift) in microservices, focusing on network, CPU, and memory utilization alongside response times. Thrift and gRPC outperformed REST based on response time and system resource efficiency, attributed to their compact binary serialization formats and efficient protocol designs.

The study [10] investigates JSONBinPack's efficiency, particularly in schema-driven mode, and directly aligns with our research. The study suggests that JSONBinPack outperforms traditional JSON and binary serialization formats based on space efficiency.

Research in [11] provides comparative experiments involving HDVM, Redis, and Protobuf for JSON data serialization, assessing performance metrics to demonstrate Protobuf's efficiency.

The paper [12] focuses on Apache Arrow and its Arrow Flight protocol. The document evaluates Apache Arrow's columnar format, leveraging it within the Arrow Flight protocol for data transfers.

The research [13] details the impact of SOAP serialization on communication efficiency, particularly in web services using HTTP and JMS protocols. The study's limitations include not considering the effect of network conditions, not testing other serialization formats like JSON or Protocol Buffers, and focusing only on SOAP messages.

Research conducted in [14] explores the efficiency of serialization formats in distributed systems, focusing on IoT devices.

The study [15] evaluates a wide range of JSON-compatible binary serialization formats. Schema-driven specifications, especially ASN.1 PER Unaligned and Apache Avro (unframed) are identified as the most space-efficient.

Additionally, for our research, it’s essential to understand the inner workings of various optimized formats, like Protocol Buffers [16], to understand better the scenarios they are suited for [17].

A recent study [18] highlighted that exploring alternative web archival formats, specifically Parquet and Avro, demonstrated significant performance improvements over the traditional WARC format.

The study [19] highlights the unique advantages of HatRPC's hint-accelerated approach in optimizing Thrift RPC services over RDMA transport.

Findings in [20] suggest that Cap’n proto is faster than Flatbuffers in serialization/deserialization time.

Source [21] suggests that MessagePack (MsgPuck) excels and beats other libraries with formats like, e.g., Flatbuffers and NanoPB (Protobuf).

Considering the issues data serialization can introduce, particularly how it might cancel out the benefits of zero-copy I/O due to CPU demands for reading, transforming, and sending message data (which leads to extra memory copies), several studies have investigated how to make this process more efficient. In several studies, it has been suggested that special hardware like FPGAs be used, as mentioned in [1].

These ideas are promising for future exploration. Yet, this study narrows its focus to exploring space efficiency for each serialization format. It sets aside these broader considerations to pinpoint the direct impact of serialization format choice.

III. SCOPE OF WORK AND OBJECTIVES

The objective of this study is to assess whether the serialized message size can be decreased by at least 30 %, compared to JSON, by utilizing alternative binary and textual serialization formats. To achieve this, we will break the objective into smaller parts. First, we will select a set of cross-platform serialization formats. Then, we will develop a real-life data model with sufficient complexity for space efficiency testing. Then, we will design a test schema for our benchmarks and convert it to corresponding schemas for each schema-based format, describing our thought process along the way. Then, we will validate the consistency of round-trip conversion for each serialization format to make sure our test is fair. Next, we will measure space efficiency for each serialization format. The last step is to perform a comparative analysis of the gathered metrics and draw conclusions.

IV. SELECTING SERIALIZATION FORMATS

In our study, we consciously decided to bypass platform-dependent serialization formats such as Java serialization, .NET Binary Formatter, and Python's Pickle due to their lack of universal compatibility and interoperability across different computing environments. While efficient within their respective ecosystems, these formats do not align with our objective of identifying serialization formats that offer broad applicability and optimal performance for inter-service communication. A combination of popularity, utility, and performance metrics guided our selection of serialization formats. We relied on indicators such as search frequencies on Google and GitHub, library download statistics, and other studies on performance evaluations focusing on serialization/deserialization speed, storage efficiency, and network bandwidth usage. This approach allowed us to identify theoretically capable formats validated through widespread practical use. Specific formats like TSF, Apache Arrow, and PSON were deliberately excluded from our analysis due to their declared usage patterns, emerging status, or lack of JVM-based tooling. However, we included JsonBinPack for specialized testing on storage efficiency, despite its limited applicability in JVM-based environments, to highlight its potential in optimizing data storage in serialization processes. Next follows the summary of some of the chosen formats.

Apache Avro is a compact, fast binary format with rich data structures and a robust, compact, and efficient serialization mechanism. It's designed to serialize data language neutrally and is often used in Apache Hadoop for big data processing. Protocol Buffers, developed by Google, are known for their simplicity and efficiency, allowing for the serialization of structured data. It's widely used in various Google internal services and external applications. Facebook originally developed Thrift, which combines a software stack with a code generation engine to build services that work efficiently and seamlessly between many languages. JsonBinPack is an efficient binary format aimed at minimizing the size of JSON documents, focusing mainly on storage efficiency, which is particularly useful for web and mobile applications where bandwidth and storage are concerns. FlatBuffers format was Developed by Google; this zero-copy format is designed for high performance with a cost of decreased memory efficiency. It allows direct access to serialized data without parsing/unpacking, making it ideal for real-time applications in specific scenarios. Cap'n Proto emphasizes speed by enabling you to access serialized data directly without parsing, like FlatBuffers. In this study, Cap'n Proto with packed encoding will be referred to as Cap'n (packed) for brevity; correspondingly, we will use Cap'n (unpacked) for encoding with no packing).

The chosen serialization formats and corresponding JVM libraries and tools used are listed in Table 1.

V. SCHEMA DESIGN

To perform a benchmark with schema-based formats, we need a test schema. We also want to ensure that the schema mirrors the complexity found in real-world payloads commonly encountered across various business scenarios. This schema should feature moderate to high levels of nesting cases. Additionally, it must be structured to allow seamless and verifiable conversion into other serialization formats like ProtoBuf or Thrift. A resulting test structure is shown in Fig. 1.

Let's take a look at the key features of the designed test schema.

The schema has several nested structures, which are good for evaluating how well a serialization format can handle hierarchical data. Nested structures, like the ones between TransactionData and AccountDetail or TransactionData and TransactionMetadata, can create depth in the data representation, challenging the serialization format to maintain efficiency.

There are various data types present, such as strings, lists, maps, sets, doubles, booleans, and custom types (like GeographicCoordinates). Testing different data types can show how the serialization format handles type encoding and whether it uses a schema-based approach (like Apache Avro) or a schema-less approach (like JSON or BSON).

There are lists (List<String>, List<Associated Document>, List<AuthorizationDetail>), maps (Map<String, Double>), and arrays (byte[]); the serialization process must deal with collections efficiently, including the overhead of item count, type information, and potential redundancy in item types.

The schema includes both composite types (like Address and AmountDetail) and primitives (like double and long). This allows for testing the serialization format's handling of flat, simple data versus complex, structured data.

The AssociatedDocument class includes a documentContent field of type byte[], which can be used to simulate the serialization of large binary data blobs. This tests the serialization format's efficiency in handling large amounts of binary data, which can significantly impact space efficiency.

Use of Enumerations: The Currency enumeration tests how the serialization format encodes symbolic data. Efficient formats may encode enums as small integers, while less efficient ones may use full-string representation.

The customAttributes field is a TreeMap<String, String>, which adds the dimension of sorted maps. It tests if the serialization format can leverage the sorted property to further optimize the data.

Each schema-based format should be adequately mapped from Test Schema, to have a representative test. Let's review several key points and insights from our mapping procedure.

For instance, Thrift required us to perform a series of transformations to and from the Test Schema as follows:

- Direct translation of Currency and TransactionType enums, leveraging native enum support in Avro and Thrift.

- Direct translation of Currency and Transaction Type enums, leveraging native enum support in Avro and Thrift.
- Complex Avro records like Geographic Coordinates, Address, AccountDetail, etc., are mapped to equivalent Thrift structs, preserving nested data structures.
- Direct mapping of Avro primitive types (e.g., string, double, long, Boolean) to Thrift's corresponding types (string, double, i64, bool).
- Avro arrays are mapped to Thrift lists (e.g., list<string>), and Avro maps to Thrift maps, maintaining collection semantics.
- Avro bytes type for binary data are directly mapped to Thrift's binary type.
- Handling Avro's nullable types via Thrift's optional field mechanism, leveraging field presence or absence to manage nullability.
- Assigning unique field identifiers in Thrift for each struct field is a requirement for Thrift's serialization and field evolution management.
- Mapping Avro long type for timestamps to Thrift's i64.

Table 1

Selected serialization formats

Format	**Library version used**
Apache Avro	1.11.0
Protocol Buffers	protobuf-java, 3.22.2
Thrift	libthrift, 0.19
JsonBinPack	npm, jsonbinpack@1.1.2
Flatbuffers	flatbuffers-java 23.5.26
Cap'n (un-packed)	java, org.capnproto.runtime 0.1.16, capnp 1.0.2
Cap'n (packed)	java, org.capnproto.runtime 0.1.16, capnp 1.0.2
MessagePack	jackson-dataformat-msgpack, 0.9.8
BSON	bson4jackson, 2.15.0
CBOR	jackson-dataformat-cbor, 2.14.2
AmazonIon	jackson-dataformat-ion, 2.14.2
Smile	jackson-dataformat-smile, 2.14.2
JSON	jackson, 2.14.2
XML	jackson-dataformat-xml, 2.14.2
YAML	jackson-dataformat-yaml, 2.14.2

The Cap'n Proto approach was similar to the one we used for Thrift; mapping from Avro to Cap'n Proto involves explicit field numbering, transforming Avro maps into Cap'n Proto lists of custom structs due to the absence of a direct map type, and assigning integer values to Enum symbols. Key differences include handling optional fields implicitly in Cap'n Proto versus Avro's nullable fields and the bytes type in Avro mapping to Data in Cap'n Proto.

For JsonBinPack, the approach follows the official benchmark. The matching schema is generated from an example JSON file. A complete testing JSON record was provided as input to the jsonbinpack compile command to create a schema. At the time of writing, there's no officially supported library for JVM-based environments, so the console tool was used instead.

Mapping between Avro and ProtoBuf schemas involves translating data types and structures to maintain semantic integrity and consistency. Basic types like string, bytes, and double are directly compatible between Avro and ProtoBuf, ensuring straightforward mappings for fields such as amount and documentContent. Compound types like arrays and maps are handled differently: Avro's array maps to Protobuf's repeated fields, and Avro's map type directly corresponds to Protobuf's map for key-value pairs. Nested structures are preserved in the translation, with nested records/messages (e.g., AccountDetail including Address, which includes GeographicCoordinates) maintaining hierarchical data relationships.

Converting the Base Schema to the FlatBuffers structure involves mapping Avro's enums directly to FlatBuffers enums, ensuring compatibility in their byte representation. Avro's primitive types and arrays translate to FlatBuffers' scalar types and vectors, respectively, requiring type alignment. Avro records, representing complex and nested structures, are converted into FlatBuffers tables, with each nested record becoming a corresponding table. Avro maps are represented as vectors of key-value pair tables in FlatBuffers, such as StringDoublePair and StringStringPair, simulating map structures by converting each map entry into instances of these tables.

Optional fields in Avro, which lack direct support for nullability in FlatBuffers, necessitate management to reflect data absence accurately. The Avro bytes type for binary data, seen in document_content, is directly converted to a [ubyte] array in FlatBuffers, ensuring proper data encoding and decoding. Timestamps represented by Avro's long type are mapped to int64 in FlatBuffers, maintaining precise time value representation. Custom attributes modeled as a map in Avro require conversion to a vector of StringStringPair in FlatBuffers.

VI. SPACE EFFICIENCY BENCHMARK

This section will analyze the space efficiency results for the selected formats and will show the reduction in comparison with JSON.

The methodology for this benchmark is as follows. The first test will evaluate space efficiency for test messages generated from epochs 1 to 30 and the second from 1 to 15. The message size will vary pseudo-randomly from ~1 kB to ~30 kB for the first test, and for the second test, the size will range from ~1 kB to ~4 kB; the size of the serialization output array will be measured and recorded. Gathered metrics will be presented as a table ordered by median serialized size in ascending order.

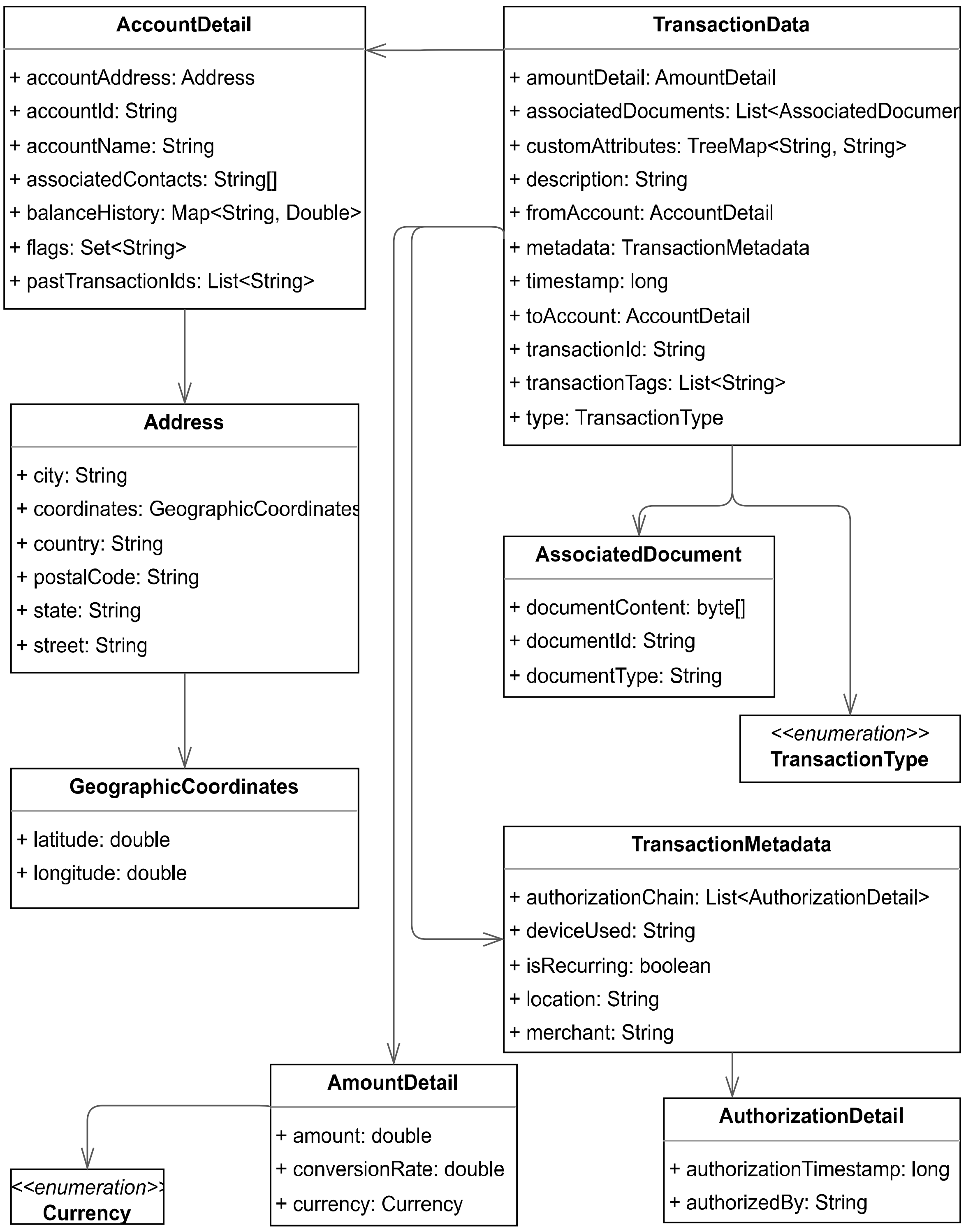

Fig. 1. A designed model for space efficiency testing

Space usage reduction in comparison with JSON (R_i) is calculated as follows:

$$R_i = \frac{S_{json} - S_i}{S_{json}} \times 100\% \qquad (1)$$

where S_{json} is the median serialized size for JSON and S_i is the median serialized size for the format in row *i*.

Let's look at our results presented in Table 2. Avro leads with a reduction of R_i=38.34 %. JsonBinPack follows at 36.81 %, with Thrift close behind at 35.97 %. This contradicts findings from [10] suggesting that JsonBinPack outperforms Avro in space efficiency.

Table 2

Serialized size over 30 epochs

Format	Mean	Median	R(%)
Avro	4871.73	2368.50	38.34
JsonBinPack	4999.93	2427.00	36.81
Thrift	5002.67	2459.50	35.97
Protobuf	5225.00	2571.00	33.06
Smile	5685.87	3016.00	21.48
Cap'n (packed)	6022.93	3020.00	21.37
MessagePack	6486.03	3250.50	15.37
CBOR	6511.77	3265.50	14.98
AmazonIon	7418.33	3747.50	2.43
JSON	7542.80	3841.00	0.00
BSON	7728.03	3974.50	-3.48
Flatbuffers	7658.40	4060.00	-5.70
Cap'n (unpacked)	7681.07	4076.00	-6.12
YAML	8251.33	4226.50	-10.04
XML	15040.80	8082.50	-110.43

Table 3

Serialized size over 15 epochs

Format	Mean	Median	R(%)
Avro	862.93	160.00	83.74
JsonBinPack	890.47	175.00	82.22
Protobuf	945.87	186.00	81.10
Thrift	932.67	217.00	77.95
Cap'n (packed)	1173.67	307.00	68.80
Smile	1403.00	631.00	35.87
Flatbuffers	1803.73	632.00	35.77
Cap'n (unpacked)	1801.07	640.00	34.96
MessagePack	1655.27	744.00	24.39
CBOR	1670.00	756.00	23.17
AmazonIon	1951.27	898.00	8.74
BSON	2073.40	954.00	3.05
JSON	2048.07	984.00	0.00
YAML	2223.53	1038.00	-5.49
XML	3788.73	1297.00	-31.81

In this case, JsonBinPack was also inferior to Avro in median space efficiency by 2.41 % and mean by 2.56 % when measured across all message variations.

Protobuf comes next, achieving a 33.06 % reduction. Smile and Cap'n (packed) show competitive efficiencies at 21.48 % and 21.37 %, respectively.

Now, let's take a look at Table 3. The results for smaller sample messages are slightly different. Avro performs significantly better in this case, with a reduction of R_i=83.74 %. Another important change is that Protobuf and Cap'n (packed) moved up the table. That might suggest that Protobuf is more efficient than Thrift in smaller message sizes. We can also see that Cap'n (packed), with a reduction of 68.8 %, significantly outperforms Smile as opposed to previous results in Table 2.

VII. CONCLUSION

Our results indicate that by switching to alternative serialization formats, it's possible to achieve more than a 30 % median reduction in serialized size compared with JSON. Specifically, utilizing the Avro format leads to size reductions of R_i=38.34 % and 83.74 % under various scenarios. Such significant reductions underscore the potential for more efficient use of network resources. By adopting these more compact formats, we can significantly decrease the amount of data transmitted over the network, potentially leading to reduced latencies and improved performance in inter-service communication.

References

[1] Marii B., Zholubak I., (2022). Features of Development and Analysis of REST Systems, *Advances in Cyber-Physical Systems*, vol. 7, no. 2, pp. 121–129, DOI: 10.23939/acps2022.02.121.

[2] Weerasinghe S., Perera I., (2024). Optimized Strategy in Cloud-Native Environment for Inter-Service Communication in Microservices, *International Journal of Online and Biomedical Engineering*, vol. 20, no. 01, pp. 40–57, DOI: 10.3991/ijoe.v20i01.44021.

[3] Proos D. P., Carlsson N., (2020). Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV, *2020 IFIP Networking Conference (Networking)*, Paris, France, pp. 10–18, [Electronic resource]. – Available at: https://ieeexplore.ieee.org/document/9142787 (Accessed: 03/22/2024).

[4] Buono V., Petrovic P., (2021). Enhance Inter-service Communication in Supersonic K-Native REST-based Java Microservice Architectures *(Dissertation)*. urn https://urn.kb.se/resolve?urn=urn:nbn:se:hkr:diva-22135

[5] Morschel L., (2020). dCache – Efficient Message Encoding For Inter-Service Communication in dCache: Evaluation of Existing Serialization Protocols as a Replacement for Java Object Serialization, *EPJ Web Conf.*, vol. 245, p. 05017, DOI: 10.1051/epjconf/202024505017.

[6] Friesel D., Spinczyk O., (2021). Data Serialization Formats for the Internet of Things, *Electronic Communications of the EASST*, vol. 20, pp. 1–4, DOI: https://doi.org/10.14279/tuj.eceasst.80.1134.

[7] Luis Á., Casares P., Cuadrado-Gallego J. J., Patricio M. A., (2021). PSON: A Serialization Format for IoT Sensor

Networks, *Sensors*, vol. 21, no. 13, p. 4559, DOI: 10.3390/s21134559.

[8] Viotti J. C., Kinderkhedia M., (2022). A Survey of JSON-compatible Binary Serialization Specifications, DOI: 10.48550/arXiv.2201.02089.

[9] Kumar P. K., Agarwal R., Shivaprasad R., Sitaram D., Kalambur S., (2021). Performance Characterization of Communication Protocols in Microservice Applications, in *International Conference on Smart Applications, Communications and Networking (SmartNets)*, pp. 1–5, DOI: 10.1109/SmartNets50376.2021.9555425.

[10] Viotti J. C., Kinderkhedia M., (2022). Benchmarking JSON BinPack, DOI: 10.48550/ARXIV.2211.12799.

[11] Huang B., Tang Y., (2021). Research on optimization of real-time efficient storage algorithm in data information serialization, *PLoS ONE*, vol. 16, no. 12, p. e0260697, DOI: 10.1371/journal.pone.0260697.

[12] Ahmad T., Ars Z. A., Hofstee H. P., (2022). Benchmarking Apache Arrow Flight - A wire-speed protocol for data transfer, querying and microservices. *arXiv*, DOI: 10.48550/arXiv.2204.03032.

[13] Dauda A. B., Adam M. S., Mustapha M. A., Mabu A. M., and Mustafa S., (2020). Soap serialization effect on communication nodes and protocols, DOI: 10.48550/ARXIV.2012.12578.

[14] Evans D., (2020). Energy-Efficient Transaction Serialization for IoT Devices, *Journal of Computer Science Research*, vol. 2, no. 2, pp. 1–16, DOI: 10.30564/jcsr.v2i2.1620.

[15] Viotti J. C., Kinderkhedia M., (2022). A Benchmark of JSON-compatible Binary Serialization Specifications, DOI: 10.48550/ARXIV.2201.03051.

[16] Protocol Buffers Version 3 Language Specification. [Electronic resource]. – Available at: https://protobuf.dev/reference/protobuf/proto3-spec/ (Accessed: 03/22/2024).

[17] Hummert, C., & Pawlaszczyk, D. (Eds.). (2022). *Mobile Forensics–The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*. Springer Nature. pp. 223–260, DOI: 10.1007/978-3-030-98467-0_9.

[18] Wang X. and Xie Z., (2020). The Case For Alternative Web Archival Formats To Expedite The Data-To-Insight Cycle, in *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020*, in JCDL '20. New York, NY, USA: Association for Computing Machinery, pp. 177–186, DOI: 10.1145/3383583.3398542.

[19] Li T., Shi H., Lu X., (2021). HatRPC: hint-accelerated thrift RPC over RDMA, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, in SC '21. New York, NY, USA: Association for Computing Machinery, pp. 1–14. DOI: 10.1145/3458817.3476191.

[20] Sorokin K., (2023). Benchmark comparing various data serialization libraries, [Electronic resource]. – Available at: https://github.com/thekvs/cpp-serializers. (Accessed: 03/22/2024).

[21] Hamerski J. C., Domingues R. P., Moraes F. G., Amory A., (2018). Evaluating Serialization for a Publish-Subscribe Based Middleware for MPSoCs, in *25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Bordeaux, France, pp. 773–776, DOI: 10.1109/ICECS.2018.8618003.

[22] Peltenburg J., Hadnagy Á., Brobbel M., Morrow R., Al-Ars Z., (2021). Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators, in *2021 ICFPT*, pp. 1–9. DOI: 10.1109/ICFPT52863.2021.9609833.

Maltsev Eduard obtained his Master's in Computer Engineering, specializing in Computer Systems and Networks, at Lviv Polytechnic National University in 2013. In 2021, he became a Certified Cloud Architect and is currently working towards a Ph.D. in Computer Engineering.

Oleksandr Muliarevych is an associate professor at the Computer Engineering Department at Lviv Polytechnic National University. He earned his PhD degree in Computer Systems and Components at Lviv Polytechnic National University in 2016.