UDC 004.045
DOI https://doi.org/10.36059/978-966-397-538-2-10

PERFORMANCE EFFICIENCY OF THE EVENT-DRIVEN DISTRIBUTED SYSTEM USING BINARY COMMUNICATION PROTOCOLS

Ph.D. S. Khlamov[1][0000-0001-9434-1081], **S. Orlov**[2][0009-0008-0680-206X], **T. Trunova**[3][0000-0003-2689-2679], **Ph.D. A. Frolov**[4][0000-0001-7335-0712], **D. Zhuzhniev**[5][0009-0001-8778-3241]

Kharkiv National University of Radio Electronics, Ukraine
EMAIL: [1]sergii.khlamov@gmail.com, [2]stasorlov21@gmail.com, [3]tetiana.trunova@nure.ua,[4]andrii.frolov@nure.ua, [5]zhuzhniev@gmail.com

Abstract*. The chapter is devoted to the performance efficiency gains from integrating binary communication protocols into event-driven distributed systems. It contends that while traditional text-based protocols like JSON provide human readability, they introduce substantial overhead in data size, serialization/deserialization speed, and network bandwidth, hindering high-traffic environments. The chapter offers a competitive analysis of binary serialization protocols including Protocol Buffers (Protobuf), MessagePack, and Apache Avro compared to text-based options such as JSON. This analysis is based on both quantitative metrics, such as serialization speed, compressed message size, and qualitative metrics like schema support, backward compatibility, and streaming vs. batch processing. The developed pipeline demonstrates that binary protocols offer substantial performance advantages, including reduced latency, increased throughput, and significant network bandwidth savings. For example, MessagePack and Protobuf are shown to achieve considerably higher serialization/deserialization speeds compared to JSON, and they also produce significantly smaller message payloads. The chapter concludes that for high-performance, low-latency, and high-throughput event-driven systems, binary protocols are frequently a fundamental prerequisite rather than merely an optimization. The chapter also offers a decision framework to assist developers and architects in choosing the appropriate protocol based on specific system requirements, with an underlying focus on ensuring the integrity of intellectual property and guarding against unauthorized duplication.*

Keywords*: Data serialization, text and binary formats, JSON, automated pipeline, performance, data serialization speed, serialized message size, network latency, data schema support and backward compatibility, event data streaming, event-driven architecture, .NET Core, MessagePack, Protobuf, Apache Avro, BenchmarkDotNet.*

Introduction

Event-Driven Architecture (EDA) inherently promotes scalability, decoupling, and resilience, making it a cornerstone for modern distributed systems. However, the

choice of communication protocol significantly influences the realization of these benefits. Traditional text-based protocols, while human-readable, introduce considerable overhead in terms of data size, serialization/deserialization speed, and network bandwidth consumption. EDA represents a fundamental shift in how distributed systems are designed and operate. Instead of direct, synchronous interactions between components, EDA orchestrates system flow through the production and consumption of events. This architectural style is increasingly prevalent in modern computing environments due to its inherent flexibility, scalability, and resilience. The core characteristics of EDA that contribute to its suitability for distributed environments include:

- *asynchronous communication:* components do not directly interact but communicate via events on the event bus;
- *decoupling and loose coupling:* event producers and consumers operate independently and are largely unaware of each other's existence;
- *scalability:* the event bus is designed to handle a high volume of events, allowing the entire system to scale horizontally;
- *resilience and fault tolerance:* events are often stored persistently in message queues or logs (e.g., Kafka, RabbitMQ).

While binary protocols undeniably enhance performance, their adoption introduces complexities related to development, debugging, and schema management. The report concludes that for high-performance, low-latency, and high-throughput event-driven systems, particularly in domains like high-frequency trading or large-scale data streaming, binary protocols are not merely an optimization but often a foundational requirement. Strategic selection and careful implementation are crucial to balance performance gains with development and operational considerations.

This article provides a competitive analysis of binary serialization protocols compared to their text alternatives. Analysis is performed by quantitative metrics (like serialization speed, compressed message size, CPU/memory consumption required for serialization process) as well as qualitative metrics (like schema support, backward compatibility, streaming vs batch processing).

The following formats are chosen during the analysis as representative of communication protocols:

- JSON is used as a lightweight data-interchange format that is easy for humans to read and write (text-based format);
- Protocol Buffer (Protobuf) is a popular binary serialization format developed by Google, emphasized by its simplicity and performance;
- MessagePack is a highly efficient binary format that represents serialized data in structures like arrays and associative arrays;
- Apache Avro is a data serialization format especially useful in big data environments and distributed systems due to its compact format, schema support, and efficient serialization.

Popular serialization protocols used by different programming languages and frameworks are outlined in Tables 1 and 2.

Table 1

Data serialization protocols compared with the typical characteristics

Format	Binary/Text	Size efficiency	Speed	Use case
JSON	Text	Large	Slow	WebAPI, REST
MessagePack	Binary	Compact	Fast	Mobile, APIs
Pickle	Binary	Medium	Fast	Python only
Protobuf	Binary	Very compact	Fast	Cross-services communication
Thrift	Binary	Compact	Fast	Microservices
Cap`n Proto	Binary	Ultra-compact	Ultra-fast	High-perf systems
Avro	Binary	Compact	Moderate	Big Data

Table 2

Data serialization protocols compared with the typical characteristics

Format	Schema required	Schema evolution	Human-readable
JSON	No	Weak	Yes
MessagePack	No	Manual	No
Pickle	No	No	No
Protobuf	Yes	Strong	No
Thrift	Yes	Moderate	No
Cap`n Proto	Yes	Safe	No
Avro	Yes	Very flexible	No

Literature review

A research paper [4] offers a valuable comparative analysis that aligns well with the article's focus on binary serialization efficiency. This study specifically investigates the impact of different serialization protocols on latency and throughput in Apache Kafka, a widely used distributed streaming platform.

The primary research goal and context primary objective is to evaluate the performance trade-offs of various serialization protocols within the context of Apache Kafka, a critical component in many event-driven and data-intensive applications.

It aims to investigate how the choice of serialization format impacts key performance metrics, such as message throughput (records per second) and latency, which are essential for maintaining high performance and scalability in streaming environments.

The research compared the following protocols: JSON, MessagePack, Protocol Buffers (Protobuf), and Apache Avro. Performance tests were conducted to measure throughput, latency, and data sizes.

In comparison, current research is not explicitly focused on any message broker or communication channel (such as Apache Kafka, REST, HTTP, or GRPC). Still, it mainly focuses on the main performance metrics of the widely used serialization protocols in general (serialization speed, message size of serialized data, CPU/memory consumption, schema support and backward compatibility).

In the meantime, the provided article's results largely corroborate the benefits of binary protocols over JSON, while also providing nuanced insights into their strengths: throughput and Latency.

Another related study [5] further supports these findings, highlighting that the compactness of serialized payloads is more critical than sheer serialization speed in reducing end-to-end latency in distributed settings.

This study also indicates that Avro, Thrift, and Protobuf exhibit more balanced performance compared to FlatBuffers and Cap'n Proto, which can underperform despite achieving high serialization speeds, suggesting that overall message size optimization is crucial for network efficiency and throughput.

While the current research provides a more general analysis of event-driven systems, the analyzed research specifically benchmarks performance within Apache Kafka, offering direct, quantifiable results for that popular streaming platform.

The analyzed research provides more detailed insights into the specific strengths of each binary protocol. In this paper, the authors classify various serialization formats (including XML, JSON, YAML, Protocol Buffers, Apache Avro, Apache Parquet, and ORC) for inter-service communication (ISC) in distributed systems, examining their historical development and current industry adoption.

The authors observe a significant industry shift toward binary formats, such as Apache Avro and Google Protocol Buffers, due to their in-deep optimization for speed and compactness.

Another paper [6] proposes an algorithm to optimize binary serialization by separating pure data from its definition, allowing for dynamic object building using

predefined templates. It highlights that binary serialization generally yields smaller times compared to JSON, especially when human-readability is not a priority.

In this research, the popular file formats (Apache Avro, Apache Parquet, JSON, and Protocol Buffers) were compared in terms of performance, ease of use, compatibility, storage efficiency, and real-world use cases for big data processing in distributed systems. It demonstrates Protobuf's efficiency in both message size and processing speed.

In the insightful article [7], the author provides an empirical analysis of widely used data streaming technologies and their associated serialization protocols, benchmarking their efficiency across various performance metrics.

The findings reveal significant performance differences and trade-offs. It highlights that while FlatBuffers and Cap'n Proto offer high serialization speeds, they may underperform in distributed settings compared to more balanced options, such as Avro, Thrift, and Protobuf, underscoring the importance of message size optimization for network efficiency.

Another article [8] focuses on analyzing and benchmarking the performance of distributed cache systems, investigating factors like the number of clients and data sizes. The study demonstrates that while factors like concurrent clients and data size significantly affect distributed cache performance, data serialization and object formats are crucial underlying implementation factors.

Data transformation mechanics: challenges and obstacles

Effective communication is the bedrock of any distributed system. The choice between text-based and binary communication protocols profoundly impacts performance efficiency. Understanding their fundamental differences is crucial for optimizing event-driven architectures.

In the comparative analysis of binary and text-based serialization formats, several key aspects warrant consideration [9]. These include schema evolution (i.e., the ability to accommodate changes in data structures over time), code generation requirements (which influence ease of integration), communication paradigms (streaming vs. batch processing), and security implications.

Text-based protocols, such as HTTP utilizing JSON, represent data in human-readable ASCII text formats.

This approach offers several advantages:

- *human readability:* data exchanged via text protocols is easily readable and interpretable by developers without requiring specialized tools, which simplifies debugging and development;
- *ease of use and wide support:* text-based formats are generally simpler to implement and enjoy broad support across various programming languages, platforms, and existing tools;
- *schema-free flexibility* (e.g., JSON): formats like JSON do not strictly require a predefined schema, offering flexibility in data structures and enabling rapid prototyping.

Despite these advantages, text-based protocols come with significant performance limitations, particularly in high-throughput or low-latency distributed systems:

- *inefficient data size:* each character in a text-based format typically consumes a single byte, leading to larger message sizes compared to binary formats, which results in higher network bandwidth consumption for data transmission;
- *slower processing:* parsing and processing text data require more CPU cycles for serialization and deserialization (data must be converted from human-readable text into a machine-readable binary form, which is a computationally intensive process);
- *overhead*: text-based payloads often include unnecessary identifiers, names, and data types that are redundant for machine processing, adding to the message size and parsing overhead.

These limitations make text-based protocols less ideal for the internal, high-volume communication characteristic of event-driven distributed systems, where raw performance and efficiency are critical.

Binary protocols encode data as sequences of bytes, representing information such as commands, identifiers, lengths, and actual data payloads directly in binary form. This approach offers inherent advantages for machine-to-machine communication:

efficiency: binary protocols are generally more efficient than text-based protocols in terms of both data size and processing speed;

- *compact data storage*: messages are significantly smaller, requiring less network bandwidth for transmission (reduction in payload size directly translates to faster data transfer);
- *faster processing*: data in binary format is closer to the machine's native language, enabling faster parsing and processing by computers, which minimizes the computational overhead associated with serialization and deserialization;
- *optimized for critical performance*: binary protocols are commonly employed in scenarios where performance and efficiency are paramount, such as internal communication within distributed systems, database query protocols, file transfer protocols, and high-performance computing environments.

While binary protocols offer clear performance superiority, it is essential to acknowledge a significant trade-off: the balance between raw performance and developer experience, including debuggability.

Binary protocols are inherently more complex to implement and debug compared to their text-based counterparts.

To facilitate this study, two representative data contracts have been defined for experimental evaluation.

The first contract is relatively simple but incorporates a variety of primitive data types, whereas the second represents a more complex structure composed of multiple nested simple objects.

The definitions of these contracts are outlined in Table 3 using C# language syntax.

Table 3

Contract definition for the serialization process

Simple data contract	Complex data contract

public class DeviceTelemetry
{
public int Id { get; set; }
public string Name { get; set; }
public bool IsActive { get; set; }
public float Temperature { get; set; }
public double Pressure { get; set; }
public DateTime Timestamp { get; set; }
public List<string> Tags { get; set; }
public Dictionary<string, int> SensorData {
get; set;
}
}
	public class Invoice
	{
	public int Id { get; set; }
	public string InvoiceNumber { get; set; }
	public DateTime InvoiceDate { get; set; }
	public DateTime DueDate { get; set; }
	public Vendor Vendor { get; set; }
	public Customer Customer { get; set; }
	public List<InvoiceItem> Details {get; set;}
	public string Notes { get; set; }
	}
	public class Vendor
	{
	public int VendorId { get; set; }
	public string CompanyName { get; set; }
	public string ContactPerson { get; set; }
	public string Address { get; set; }
	public string Email { get; set; }
	public string Phone { get; set; }
	public string TaxNumber { get; set; }
	public string BankAccount { get; set; }
	}
	public class Customer
	{
	public int CustomerId { get; set; }
	public string CompanyName { get; set; }
	public string ContactPerson { get; set; }
	public string Address { get; set; }
	public string Email { get; set; }
	public string Phone { get; set; }
	}
	public class InvoiceItem
	{
	public string Description { get; set; }
	public int Quantity { get; set; }
	public decimal UnitPrice { get; set; }
	public decimal TaxRate { get; set; }
	}

The *DeviceTelemetry* class defines a structured data model representing telemetry information generated by a device. It encapsulates a combination of scalar, collection, and complex types, making it suitable for evaluating serialization strategies across a range of data patterns.

Its properties describe device identification, operational status, current temperature/pressure parameters, the exact time at which the telemetry data was captured, and the collection of descriptive tags, metadata, and sensor identifiers.

The *Invoice* class defines a comprehensive business document model used to represent billing information in transactional systems. It is composed of nested complex types and collections, making it suitable for evaluating serialization performance in hierarchical and object-rich data structures.

Its data encapsulates the overall invoice data, including the organization or individual issuing the invoice, the buyer or recipient of the invoice, and defines the individual line items billed on the invoice.

Multiple contracts are provided to analyze and compare serialization for objects of varying sizes and data types (custom nested properties).

1.1 Text-based serialization process

Text serialization format uses human-readable text to store and transmit data objects consisting of "name–value" pairs and arrays (or other serializable values). It is a commonly used data format with diverse applications in electronic data interchange, including web applications and server interactions.

For example, JSON (JavaScript Object Notation), the most popular text serialization process, converts data structures or objects from program memory into a JSON-formatted string [10].

This string represents a human-readable and lightweight text-based format in two primary structures:

1. **Objects (key-value pairs):** unordered sets of key/value pairs.

Keys are strings, and values can be a string, a number, a boolean (true/false), null, an array, or another JSON object.

In programming languages, this typically maps to dictionaries, hash maps, or objects. An example of a JSON object:

{"name": "Sensor − X9","temperature": 36.5}

2. **Arrays (ordered lists):** ordered sequences of values.

Values can be any valid JSON data type. In programming languages, this maps to lists or arrays.

An example of a JSON array:

["env", "critical", "zone−1"]

When you serialize a data structure (e.g., a Python dictionary, a Java object, a JavaScript object) into JSON, the following conceptual steps occur:

1. **Data types mapping**: the serializer maps the native data types of the programming language to their corresponding JSON types:

a. Numbers (integers, floats) -> JSON numbers;

b. Strings -> JSON strings (often requiring proper escaping of special characters like quotes, backslashes, etc.);

c. Boolean (true/false) -> JSON true/false;
d. Null/None -> JSON null;
e. Lists/Array -> JSON arrays;
f. Dictionaries/Objects -> JSON objects.

2. **Conversion to string**: the data structure is traversed, and each piece of data is converted into its JSON string representation. Keys and string values are enclosed in double quotes. Objects are enclosed in curly braces {}, and arrays in square brackets [].

3. **Concatenation**: the individual string representations are concatenated to form a single, continuous JSON string.

To perform serialization of the simple data contract defined before, let's initialize its properties with sample data to understand the destination object.

Figure 1 contains the outcome of the contract definition using the C# programming language.

```
var telemetry = new DeviceTelemetry
{
    Id = 101,
    Name = "Sensor-X9",
    IsActive = true,
    Temperature = 36.5f,
    Pressure = 1013.25,
    Timestamp = new DateTime(2025, 7, 1, 15, 30, 0, DateTimeKind.Utc),
    Tags = ["env", "critical", "zone-1"],
    SensorData = new Dictionary<string, int>
    {
        ["humidity"] = 45,
        ["vibration"] = 7
    }
};

return telemetry;
```

Figure 1. Initialized contract definition with different data types using the C# programming language

The random values were used during the object initialization in the C# programming language, and we achieved the following initialized object.

The outcome of the JSON serialization process would be a JSON object containing a text representation of the C# class definition presented in Figure 2.

While transferring through the network, each character is represented by an array of bytes in UTF-8 notation. A C# command to serialize an object into JSON format is provided in Figure 3. Invoking the JSON serialization by the following command in C# language, we would receive its representation in the JSON format. To analyze the JSON string from a byte array perspective, we should convert each character into its respective byte. And that will be the actual size of the serialized message transferred over the network. For example, the curly bracket (`{`) represents a 7B UTF-8 hex string. 22 code is used for the quotation (“) mark, and 49 code is used for the “I” character.

```
{
    "Id": 101,
    "Name": "Sensor-X9",
    "IsActive": true,
    "Temperature": 36.5,
    "Pressure": 1013.25,
    "Timestamp": "2025-07-01T15:30:00Z",
    "Tags": [
        "env",
        "critical",
        "zone-1"
    ],
    "SensorData": {
        "humidity": 45,
        "vibration": 7
    }
}
```


Figure 2. String representation of DeviceTelemetry object serialized in JSON format

```
internal class JsonSerializer
{
    private static readonly JsonSerializerSettings jsonSerializerSettings;
    private static readonly Newtonsoft.Json.JsonSerializer jsonSerializer;

    static JsonSerializer()
    {
        jsonSerializerSettings = new JsonSerializerSettings
        {
            ConstructorHandling = ConstructorHandling.AllowNonPublicDefaultConstructor,
            ContractResolver = new PrivateContractResolver()
        };

        jsonSerializer = Newtonsoft.Json.JsonSerializer.Create(jsonSerializerSettings);
    }

    public static byte[] SerializeToUtf8Bytes<T>(T instance)
    {
        using StringWriter stringWriter = new();
        jsonSerializer.Serialize(stringWriter, instance);

        return Encoding.UTF8.GetBytes(stringWriter.GetStringBuilder().ToString());
    }

    public static T Deserialize<T>(byte[] buffer)
    {
        JsonReader jsonTextReader = new JsonTextReader(new StringReader(Encoding.UTF8.GetString(buffer)));

        T? entity = jsonSerializer.Deserialize<T>(jsonTextReader);

        return entity is null ? throw new Exception("JSON serialization exception") : entity;
    }
}
```

Figure 3. JSON serialization logic

Keeping each character as a representation of an array of bytes, we will ultimately arrive at the output displayed in Figure 4.

If we calculate the total number of bytes for the resulting message, we will achieve a total of 230 bytes. This is our initial point for analyzing binary serialization formats compared to text-based ones.

7B-22-49-64-22-3A-31-30-31-2C-22-4E-61-6D-65-22-3A-22-53-65-6E-73-6F-72-2D-58-39-22-2
C-22-49-73-41-63-74-69-76-65-22-3A-74-72-75-65-2C-22-54-65-6D-70-65-72-61-74-75-72-65-2
2-3A-33-36-2E-35-2C-22-50-72-65-73-73-75-72-65-22-3A-31-30-31-33-2E-32-35-2C-22-54-69-
6D-65-73-74-61-6D-70-22-3A-22-32-30-32-35-2D-30-37-2D-30-31-54-31-35-3A-33-30-3A-30-3
0-5A-22-2C-22-54-61-67-73-22-3A-5B-22-65-6E-76-22-2C-22-63-72-69-74-69-63-61-6C-22-2C-
22-7A-6F-6E-65-2D-31-22-5D-2C-22-53-65-6E-73-6F-72-44-61-74-61-22-3A-7B-22-68-75-6D-6
9-64-69-74-79-22-3A-34-35-2C-22-76-69-62-72-61-74-69-6F-6E-22-3A-37-7D-7D

Figure 4. Binary representation of a serialized JSON object

1.2 MessagePack serialization process

MessagePack serialization format is a computer data interchange format. It is a binary format for representing simple data structures, such as arrays and associative arrays. MessagePack aims to be as compact and straightforward as possible [3].

MessagePack supports a variety of data types, like JSON, but encodes them directly into binary:

- **Numbers:** Integers and floating-point numbers are encoded efficiently using fixed-size integer types (uint8, int64) or floating-point types (float32, float64). Smaller numbers use fewer bytes;
- **Strings** encoded with a length prefix followed by the UTF-8 encoded bytes of the string;
- **Booleans** encoded as single bytes (0xc3 for true, 0xc2 for false);
- **Null** encoded as a single byte (0xc0);
- **Arrays** encoded with a type of marker indicating the array size, followed by the serialized elements of the array;
- **Maps (Objects)** encoded with a type of marker indicating the map size, followed by interleaved serialized key-value pairs. Keys and values can be any MessagePack type.

Where a data structure is serialized using the MessagePack binary format, the following conceptual steps occur during the serialization process [3]:

1.**Type identification and header/marker**: the serializer identifies the data type (e.g., integer, string, array, map). Based on the type and often its size, it writes a small type of marker byte (or bytes) that signals the type of data that follows and sometimes its length. For instance, there are markers for "positive fixint" (a single byte for small integers), "fixstr" (a byte indicating the string length, up to 31 bytes), or "map 16" (indicating a 16-bit length for a map);

2.**Data encoding**: the actual data is then encoded directly into binary following the marker;

3.**Numbers**: integers are stored as raw binary bytes (e.g., a uint32 would take 4 bytes);

4.**Strings**: the length is written, followed by the UTF-8 bytes of the string;

5.**Arrays/maps**: the size is written, then the serializer recursively processes each element (for arrays) or key-value pair (for maps) until the entire structure is converted;

6.**Byte stream generation**: all the encoded bytes are appended to form a contiguous binary stream.

MessagePack format converts data into a sequence of bytes, reducing the total sequence size. An efficient encoding scheme assigns data types and their respective values to bytes in a very compact manner.

MessagePack is a schema-less binary serialization format by default, meaning it does not require or embed an explicit schema like Protobuf or Avro.

As a result, the byte representation of the serialized DeviceTelemetry message can be found in Figure 5.

98-65-A9-53-65-6E-73-6F-72-2D-58-39-C3-CA-42-12-00-00-CB-40-8F-AA-00-00-00-00-00-D6-FF-68-63-FE-F8-93-A3-65-6E-76-A8-63-72-69-74-69-63-61-6C-A6-7A-6F-6E-65-2D-31-82-A8-68-75-6D-69-64-69-74-79-2D-A9-76-69-62-72-61-74-69-6F-6E-07

Figure 5. Binary representation of serialized DeviceTelemetry object in MessagePack format

As mentioned, serializing the provided object in the MessagePack format significantly reduces the resulting message size. The MessagePack format converts data into a sequence of bytes, thereby reducing the total sequence size.

An efficient encoding scheme assigns data types and their respective values to bytes in a very compact manner. We can think of it as an array with the sequence number, where each byte is held in a special position [2].

Table 4 explains each byte and how the compression works.

Table 4

Explanation of each message pack serialized byte

Byte representation	Property	Explanation
98	Declare a fixed array of 8 elements	MessagePack format represents the data as an array of fixed size
65	Id: int 101	65 → 101
A9 53 65 6E 73 6F 72 2D 58 39	Name: "Sensor-X9"	str(9) A9 53 65 6E 73 6F 72 2D 58 39 = "Sensor-X9"
C3	IsActive: true	C3 → true
CA 42 0E 00 00	Temperature: 36.5f (float32)	CA 42 0E 00 00 = 36.5
CB 40 8F 4A 00 00 00 00 00	Pressure: 1013.25 (float64)	CB 40 8F 4A 00 00 00 00 00 = 1013.25
D6 FF 00 00 01 91 4B D8 10	Timestamp: ext 8 (DateTime)	D6 FF (ext type -1) + 8-byte epoch timestamp
93 A3 65 6E 76 A8 63 72 69 74 69 63 61 6C A7 7A 6F 6E 65 2D 31	Tags: ["env", "critical", "zone-1"]	array of 3 strings (Tags)
82 A8 68 75 6D 69 64 69 74 79 2D 34 35 A9 76 69 62 72 61 74 69 6F 6E 07	"SensorData": { "humidity": 45, "vibration": 7 }	map of 2 entries (SensorData)

If we calculate the total number of bytes for the resulting message, we will obtain a total of 84 bytes. Therefore, the MessagePack serialization format reduces the message size compared to the test JSON format by approximately 3 times.

1.3 Protocol Buffers serialization process

Protocol Buffers (Protobuf) serialization is a language-agnostic, platform-neutral, and extensible mechanism for serializing structured data. Unlike JSON or XML, Protobuf serializes data into a compact binary format, making it highly efficient in terms of size and speed. Its "schema-first" approach is central to its mechanism [11].

Before you can serialize or deserialize data with Protobuf, you must define the structure of your data using a simple Interface Definition Language (IDL) in the ".proto" file. This scheme specifies:

1. **Message types**: the structure of your data, analogous to classes or structures;
2. **Fields**: each field within a message has:

a. Type: int32, string, bool, bytes, or another message type;

b. Name: product_id, name;

c. Unique tag number is crucial for identifying fields in the binary format and enabling;

d. schema evolution (e.g., 1, 2, 3);

e. Rule: optional, required - though required is generally discouraged in modern Protobuf for schema evolution reasons, and repeated for lists;

3. **Field iteration**: the serializer iterates through each field in the message object that has a value (default values are not serialized to save space);
4. **Key-Value pairs** (Wire Format): for each field, Protobuf encodes a "key-value" pair in the binary stream. The "key" is a combination of the field's unique tag number and its wire type;

a. Tag Number: directly from the ".proto" file (e.g., 1, 2, 3);

b. Wire Type: indicates the data type of the field on the wire, telling the parser how to interpret the value that follows. Common wire types: Varint, Fixed64, Length-delimited, Fixed32;

5. **Value encoding**: the field's actual value is then encoded according to its wire type.

a. Varints: integers are encoded using a variable-length encoding, where smaller numbers occupy fewer bytes. This is efficient for typical integer values;

b. Length-delimited: for strings, Protobuf first writes the length of the string as a varint, then writes the UTF-8 bytes of the string itself. Similarly, for byte fields or nested messages;

c. Fixed-size: fixed-size integers and floating-point numbers are written directly as a fixed number of bytes (e.g., 4 bytes for float, 8 bytes for double);

6. **Concatenation**: all encoded field key-value pairs are concatenated into a single binary byte stream. The order of fields in the proto file doesn't strictly dictate the order in the binary stream and fields can be serialized in any order.

Protocol Buffers (Protobuf) uses a ".proto" file as a schema definition file. It's used to define the structure of your data in a language-neutral, platform-neutral, and backward-compatible format [2].

This file describes the messages (data types) that your app will send and receive, including the fields they contain, such as types, names, and field numbers.

The proto equivalent of the DeviceTelemetry contract (Protobuf schema) is displayed below in Figure 6.

```
syntax = "proto3";

message DeviceTelemetry {
  int32 id = 1;
  string name = 2;
  bool isActive = 3;
  float temperature = 4;
  double pressure = 5;
  int64 timestamp = 6;  // Unix time in seconds
  repeated string tags = 7;
  map<string, int32> sensorData = 8;
}
```

Figure 6. Protobuf file representing the schema for the DeviceTelemetry object

By invoking serialization using the Protobuf format, we would receive the sequence of bytes shown in Figure 7.

08-65-12-09-53-65-6E-73-6F-72-2D-58-39-18-01-25-00-00-12-42-29-00-00-00-00-00-AA-8F-40-32-07-08-C4-99-EB-1B-10-02-3A-03-65-6E-76-3A-08-63-72-69-74-69-63-61-6C-3A-06-7A-6F-6E-65-2D-31-42-0C-0A-08-68-75-6D-69-64-69-74-79-10-2D-42-0D-0A-09-76-69-62-72-61-74-69-6F-6E-10-07

Figure 7. Binary representation of the serialized DeviceTelemetry object in Protobuf format

As mentioned, serializing the provided object in the Protobuf format reduces the resulting message size, as does the MessagePack format. Canonically, messages are serialized into a compact binary wire format that is forward- and backward-compatible, but not self-describing. Unfortunately, there is no way to tell the names, meaning, or full datatypes of fields without an external specification [2].

Table 5 explains each byte and how the compression works. If we calculate the total number of bytes for the resulting message, we will achieve a total of 98 bytes.

Table 5

Explanation of each Protobuf serialized byte

Byte representation	Property	Explanation
08 65	Id = 101	Field 1 (varint): 8 → id, 101 = 0x65
12 09 "Sensor-X9"	Name	Field 2 (length-prefixed): 9 bytes
18 01	IsActive	Field 3 (bool): 1 = true
25 00 00 12 42	emperature	Field 4 (float): 36.5 in IEEE 754
31 00 00 8F 40 1C DD 40 40	Pressure	Field 5 (double): 1013.25 in IEEE 754
38 00 AC 86 66	imestamp	Field 6 (varint): 1756990200 = Unix time
3A 03 65 6E 76	Tags[0]	Field 7: length-delimited string "env"
3A 08 63 72 69 74 69 63 61 6C	Tags[1]	Field 7: "critical"
3A 07 7A 6F 6E 65 2D 31	Tags[2]	Field 7: "zone-1"

Therefore, we can see that the Protobuf serialization format reduces the message size compared to the test JSON format by approximately 3 times, like MessagePack.

1.4 Apache Avro serialization process

Apache Avro uses a compact binary format for data serialization, accompanied by a separate (or embedded) JSON-based schema [3].

Apache Avro requires a JSON schema to define the data structure [3]. This schema is either embedded in the file or is provided separately during serialization/deserialization (common in RPC or Kafka contexts). Avro uses a binary encoding that provides compact size, fast processing, and cross-language support [2].

This scheme describes the structure of the DeviceTelemetry class (Figure 8).

Avro format supports a variety of data types for efficient encoding and decoding [13]:

- Int - Variable-length encoded signed integer (uses fewer bytes for small numbers);
- Long - encoded 64-bit integer;
- Float - 4 bytes (IEEE 754);
- Double - 8 bytes (IEEE 754);
- Boolean - 1 byte: 0x00 = false, 0x01 = true;
- String - UTF-8 encoded with variable-length prefix for byte count;
- Array - Block-encoded: [block-count][item...][0];
- Map - Block-encoded like array: [count][key][value]...[0];
- Record - Fields encoded in schema-defined order.

```
{
  "type": "record",
  "name": "DeviceTelemetry",
  "namespace": "Example",
  "fields": [
    { "name": "Id", "type": "int" },
    { "name": "Name", "type": "string" },
    { "name": "IsActive", "type": "boolean" },
    { "name": "Temperature", "type": "float" },
    { "name": "Pressure", "type": "double" },
    { "name": "Timestamp", "type": { "type": "long", "logicalType": "timestamp-millis" } },
    { "name": "Tags", "type": { "type": "array", "items": "string" } },
    {
      "name": "SensorData",
      "type": {
        "type": "map",
        "values": "int"
      }
    }
  ]
}
```

Figure 8. Avro JSON schema for the structure definition

Invoking the serialization using an Avro format, we would receive the sequence of bytes provided in Figure 9.

```
CA 01 12 53 65 6E 73 6F 72 2D 58 39 01 00 00 12 42 00 00 00 00 00 44 8F 40
80 F7 D5 E2 FB 2C 06 06 65 6E 76 10 63 72 69 74 69 63 61 6C 0E 7A 6F 6E 65
2D 31 00 04 10 68 75 6D 69 64 69 74 79 5A 0E 76 69 62 72 61 74 69 6F 6E 0E 00
```

Figure 9. Binary representation of the serialized DeviceTelemetry object in Avro format

Table 5 explains each byte and how the compression works.

Table 5

Explanation of each Apache Avro serialized byte

Byte representation	Property	Explanation
ca 01	Id = 101	ZigZag + Varint (101 × 2 = 202)
12 53 65 6E 73 6F 72 2D 58 39	Name	"Sensor-X9" Length=9, then UTF-8 bytes
01	IsActive	Field 3 (bool): 1 = true
00 00 12 42	Temperature	Field 4 (float): 36.5 in IEEE 754
00 00 00 00 00 44 8f 40	Pressure	Field 5 (double): 1013.25 in IEEE 754
80 f7 d5 e2 fb 2c	Timestamp	Varint encoded 64-bit long
06 06 65 6e 76 10 63 72 69 74 69 63 61 6c 0e 7a 6f 6e 65 2d 31 00	Tags	Block count + strings + end
04 10 68 75 6d 69 64 69 74 79 5a 0e 76 69 62 72 61 74 69 6f 6e 0e 00	SensorData	Map block: key1 + val1, key2 + val2
ca 01	Id = 101	ZigZag + Varint (101 × 2 = 202)

If we calculate the total number of bytes for the resulting message, we will obtain a total of 79 bytes.

Therefore, we can see that Apache Avro serialization format reduces the message size compared to the test JSON format by approximately 3 times, like MessagePack. Furthermore, Apache Avro is more efficient than Message Pack and Protobuf serializers [12].

Automation pipeline for the performance metrics

4.1. System architecture

The system architecture for the performance pipeline of binary serializers is constructed using a combination of modern technologies and frameworks to ensure high performance, scalability, and ease of deployment [13].

The architecture is built on the .NET Core framework using the C# programming language and the BenchmarkDotNet package. BenchmarkDotNet is the de facto benchmarking library for .NET. It enables you to accurately and reliably measure and compare the performance (speed, memory usage, etc.) of your C# code using micro-benchmarking techniques. It is ideally suited for open-source projects, allowing the development of optimized algorithms for serialization, speed, and memory comparison. It provides the following features for precise measurements: high precision (utilizing multiple runs, warm-up, GC, and statistics), multiple exporters (Markdown, CSV, HTML, and JSON), and memory diagnostics.

4.2 Automation pipeline

Using a GitHub Actions pipeline with BenchmarkDotNet for benchmarking provides several practical benefits, particularly in CI/CD, regression detection, and performance tracking. The following benefits are provided by the GitActions

pipeline for performance measurements: performance regression detection, consistent and repeatable benchmarks, automation, tracking performance over time, and no local dependencies. The workflow runs on every push to the main branch and, optionally, on pull requests, using Ubuntu-Latest as the execution environment.

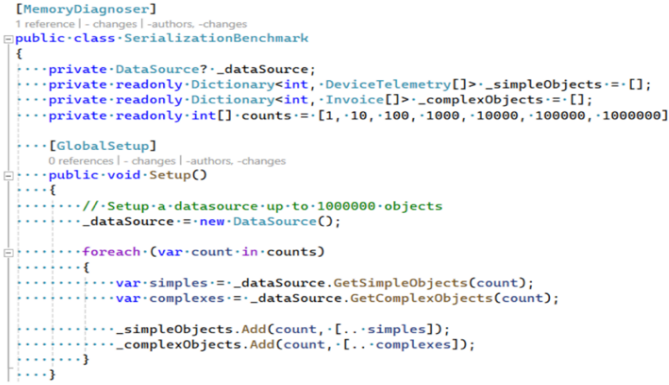
During the run, it executes the BenchmarkDotNet project via dotnet run -c Release, targeting the specific .csproj that contains serializer performance tests. Artifacts are saved to BenchmarkDotNet.Artifacts/results/ in formats such as Markdown (.md), JSON, CSV, and HTML. Running benchmarks in CI avoids variability across developer machines. Storing artifacts allows tracking performance over time or comparing past runs. Figure 10 provides an implementation of the GitActions pipeline.

The workflow runs on every push to the main branch and optionally on pull requests and uses ubuntu-latest as the execution environment. During the run, it executes the BenchmarkDotNet project via dotnet run -c Release, targeting the specific .csproj that contains serializer performance tests. Artifacts are saved to BenchmarkDotNet.Artifacts/results/ in formats like Markdown (.md), JSON, CSV, and HTML. Running benchmarks in CI avoids variability across developer machines. Storing artifacts allows tracking performance over time or comparing past runs.

```yaml
name: Benchmark.Net
on:
  push:
    branches:
      - master

permissions:
  contents: write
  deployments: write

jobs:
  benchmark:
    name: Run benchmark
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-dotnet@v3
        with:
          dotnet-version: '8.0.x'  # SDK Version to use. keep in line with
examples/benchmarkdotnet/global.json

      - name: Restore Dependencies
        run: cd src/BinarySerializers && dotnet restore

      - name: Run Benchmark & Serialization Compression reports
        run: cd src/BinarySerializers && dotnet run -c Release --exporters json --
filter '*'

      - name: Extract Serialization Compression Results
        id: find_compression_report
        run: |
          mkdir temp
          REPORT=$(find . -type f -name 'serialization_compression.md' | head -n 1)
          echo "compression_file=$REPORT" >> $GITHUB_OUTPUT

      - name: Show compression report
        run: |
          echo "Selected compression report: ${{
steps.find_compression_report.outputs.compression_file }}"

      - name: Copy compression report
        run: |
          cp "${{ steps.find_compression_report.outputs.compression_file }}"
temp/compression.md

      - name: Store compression result in README.md
        run: |
          start_marker="<!-- COMPRESSION_START -->"
          end_marker="<!-- COMPRESSION_END -->"
          temp_file="temp/README.new"

          awk -v s="$start_marker" -v e="$end_marker" '
            BEGIN {in_section=0}
            $0 ~ s {print; print ""; while ((getline line < "temp/compression.md") >
0) print line; in_section=1; next}
            $0 ~ e {in_section=0}
            !in_section {print}
          ' README.md > "$temp_file"

          mv "$temp_file" README.md

      - name: Commit README.md
        run: |
          git config user.name "github-actions[bot]"
          git config user.email "github-actions[bot]@users.noreply.github.com"
          git add README.md
          git commit -m "Update README from Compression size report [skip ci]" ||
echo "No changes to commit"

      - name: Extract Benchmark Results
        id: find_benchmark_report
        run: |
          REPORT=$(find . -type f -name '*-report-github.md' | head -n 1)
          echo "report_file=$REPORT" >> $GITHUB_OUTPUT

      - name: Show selected report
        run: |
          echo "Selected benchmark report: ${{
steps.find_benchmark_report.outputs.report_file }}"

      - name: Copy benchmark report
        run: |
          cp "${{ steps.find_benchmark_report.outputs.report_file }}"
temp/benchmark.md

      - name: Debug benchmark.md content
        run: |
          echo "Contents of benchmark.md:"
          cat temp/benchmark.md

      - name: Store benchmark result in README.md
        run: |
          start_marker="<!-- BENCHMARK_START -->"
          end_marker="<!-- BENCHMARK_END -->"
          temp_file="temp/README.new"

          awk -v s="$start_marker" -v e="$end_marker" '
            BEGIN {in_section=0}
            $0 ~ s {print; print ""; while ((getline line < "temp/benchmark.md") >
0) print line; in_section=1; next}
            $0 ~ e {in_section=0}
            !in_section {print}
          ' README.md > "$temp_file"

          mv "$temp_file" README.md

      - name: Commit and push README.md
        run: |
          git config user.name "github-actions[bot]"
          git config user.email "github-actions[bot]@users.noreply.github.com"
          git add README.md
          git commit -m "Update README from BenchmarkDotNet report [skip ci]" ||
echo "No changes to commit"
          git push
```

Figure 10. GitActions pipeline implementation for the performance metrics

The benchmarking cases are implemented in the best-practice way, following Microsoft recommendations, which can be found in Figure 11.



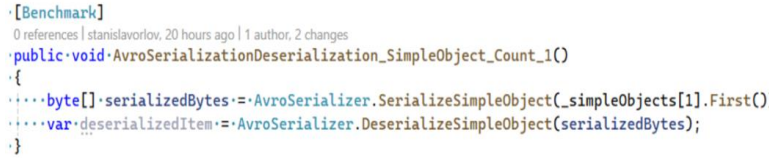
```
[MemoryDiagnoser]
1 reference | - changes | -authors, -changes
public class SerializationBenchmark
{
    private DataSource? _dataSource;
    private readonly Dictionary<int, DeviceTelemetry[]> _simpleObjects = [];
    private readonly Dictionary<int, Invoice[]> _complexObjects = [];
    private readonly int[] counts = [1, 10, 100, 1000, 10000, 100000, 1000000];

    [GlobalSetup]
    0 references | - changes | -authors, -changes
    public void Setup()
    {
        //Setup a datasource up to 1000000 objects
        _dataSource = new DataSource();

        foreach (var count in counts)
        {
            var simples = _dataSource.GetSimpleObjects(count);
            var complexes = _dataSource.GetComplexObjects(count);

            _simpleObjects.Add(count, [..simples]);
            _complexObjects.Add(count, [..complexes]);
        }
    }
```

Figure 11. Benchmark case initialization with test data provided

This C# snippet defines a benchmarking class using BenchmarkDotNet to measure memory and performance characteristics of serialization operations. [MemoryDiagnoser] attribute from BenchmarkDotNet enables memory usage diagnostics during benchmarks (e.g., allocated bytes, GC collections). Essential for measuring serialization performance in memory-sensitive applications.

The setup method runs once before all benchmarks and prepares similar test data at different scales. Pipeline benchmarking different serializers (e.g., JSON, Protobuf, MessagePack), analyzes how performance scales with object count and compares GC (memory) pressure across serializers.

The benchmark snippet, located in Figure 12, demonstrates how the test performs serialization and deserialization, and measures performance metrics.



```
[Benchmark]
0 references | stanislavorlov, 20 hours ago | 1 author, 2 changes
public void AvroSerializationDeserialization_SimpleObject_Count_1()
{
    byte[] serializedBytes = AvroSerializer.SerializeSimpleObject(_simpleObjects[1].First());
    var deserializedItem = AvroSerializer.DeserializeSimpleObject(serializedBytes);
}
```

Figure 12. Simple benchmark for message serialization using Avro format

In addition, it provides performance metrics (serialization speed and obtained compression size) required for the comparison report.

Performance efficiency metrics and impact of binary serialization protocols

The tables below illustrate the results obtained during pipeline execution and summarize the metrics obtained during execution. We can see the different objects used for serialization: flat and nested objects.

The difference relies on the results, message size and object complexity for the serialization. “Flat contract” means the simple objects without any nested objects

inside, with a couple of properties of different data types. While the "Nested objects in contract" means the usage of multiple "Flat contracts" inside a single object. Meaning the serialization was done on the top level and all the lower levels.

From Table 6, we can see the metrics on how many binary serializers (Message Pack, Protobuf, and Apache Avro) work faster and have a smaller compression size than a simple text formatter (JSON). Figure 13 shows the diagram illustrating the difference in serialization speed between formats.

Figure 13. Difference in the serialization speed between different formats

Table 6

Serialization comparison speed and size of binary serializers compared to Text (JSON)

Number of objects	MessagePack		Protobuf		Apache Avro	
Flat contract						
	Speed	Size	Speed	Size	Speed	Size
1	5.3x	2.7x	5.6x	2.3x	1.7x	3x
10	5.3x	2.7x	7.1x	2.3x	1.6x	2.9x
100	5.1x	2.7x	6.6x	2.3x	1.4x	2.9x
1000	5.5x	2.7x	6.9x	2.3x	1.4x	2.9x
10000	5.8x	2.7x	8.3x	2.3x	1.2x	2.9x
100000	5.0x	2.7x	8.9x	2.3x	1.3x	2.9x
1000000	4.6x	2.7x	9.6x	2.3x	1.4x	2.9x
Nested objects in a contract						
	Speed	Size	Speed	Size	Speed	Size
1	3.3x	2.5x	9.3x	2.2x	1x	2.5x
10	3.4x	2.4x	12.6x	2.2x	1x	2.5x
100	3.5x	2.5x	13.7x	2.2x	0.9x	2.5x
1000	3.5x	2.5x	13.5x	2.2x	0.9x	2.5x
10000	3.2x	2.5x	14.2x	2.2x	0.7x	2.5x
100000	3.2x	2.5x	14.6x	2.2x	0.8x	2.5x
1000000	2.5x	2.5x	15.7x	2.2x	0.7x	2.5x

Conclusions

These protocols generate significantly smaller message payloads, resulting in reduced network bandwidth usage and lower operational costs. Furthermore, they require fewer CPU cycles for processing, resulting in much faster data handling and lower latency [3]. Despite their clear performance superiority, the adoption of binary protocols introduces certain complexities, including development overhead, debugging challenges, and schema management.

Ultimately, the decision to adopt a binary serialization protocol should be guided by a clear understanding of the system's specific requirements. Developers and architects are encouraged to utilize a decision matrix or a set of guiding questions, considering factors such as latency requirements (ms, µs, ns), expected message volume/throughput, acceptable developer overhead, learning curve, and existing tooling/ecosystem considerations.

By carefully weighing these factors, organizations can effectively harness the power of binary serialization to build highly efficient, scalable, and performant distributed systems.

Choosing the proper serialization protocol is crucial for balancing performance, development speed, and maintainability in distributed systems [15]. This framework helps you weigh the key factors:

- *Performance requirements (latency & throughput)*: binary protocols minimize the message size and maximize serialization/deserialization speed; text protocols are likely sufficient.
- *Schema evolution & compatibility*: Protobuf and Avro are designed with robust schema evolution mechanisms. They ensure that older and newer versions of services can communicate seamlessly as your data structures evolve. MessagePack's ability to ignore unknown fields provides forward compatibility. JSON's schema-less nature means schema evolution is managed at the application level.
- *Tooling & ecosystem integration*: Protobuf is the native choice for gRPC, offering highly optimized inter-service communication. Acro is a de facto standard in the Apache ecosystem, with excellent integration with Kafka. Both MessagePack and JSON have broad language support. MessagePack is great for compact messages in constrained environments (IoT). JSON is universal for web APIs.

In essence, if your system's performance (latency and throughput) is a critical bottleneck, invest in binary serialization. If development speed, human readability, and simpler tooling are paramount, and performance needs are moderate, text-based formats might be perfectly adequate.

The optimal choice often involves a trade-off between raw performance and development/operational complexity. Future work should explore the integration of binary communication protocols with containerized microservices orchestrated in cloud-native environments, such as Kubernetes-based astronomical pipelines for knowledge discovery in databases [16, 17] and data mining [18].

Specifically, the compatibility of high-efficiency protocols like Protocol Buffers or FlatBuffers with real-time processing frameworks (e.g., Apache Flink or Dask) in

astronomical data streams [19] needs further evaluation. Additional attention should be given to fault tolerance and protocol resiliency in high-throughput settings, mainly when operating across intermittent or high-latency network links, such as those connecting remote observatories and telescopes with centralized data centers [20].

Another promising research direction lies in adapting these protocols for the control systems of robotic telescopes and autonomous observatories, where event-driven communication enables coordinated scheduling, dynamic resource allocation, and the triggering of specialized hardware such as adaptive optics systems [21] or spectroscopic instruments.

Furthermore, binary protocols could enhance distributed machine learning [22] frameworks used for astronomical image classification [23, 24] and anomaly detection [25, 26], by reducing communication bottlenecks during distributed training or model inference. The application of this research to edge AI models deployed near the data source opens new pathways for intelligent in-sensor processing [27], supporting decision-making processes [28], and rapid, autonomous scientific discovery [29] in the next generation of sky surveys [30].

References

1. Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed systems: Concepts and design* (5th ed., 1080 p.). Pearson.

2. Grigorik, I. (2011). *Protocol Buffers, Avro, Thrift & MessagePack*. Retrieved from https://www.igvita.com/2011/08/01/protocol-buffers-avro-thrift-messagepack

3. GitHub Repository. (2025). *Benchmarks against serialization systems*. Retrieved from https://github.com/saint1991/serialization-benchmark

4. Myastovskiy, T. (2024). *Evaluating the performance of serialization protocols in Apache Kafka* (Report). Umeå University. Retrieved from https://www.diva-portal.org/smash/get/diva2:1878772/FULLTEXT01.pdf

5. Maltsev, E., Muliarevych, O., & Razzaque, A. (2024). Classifying serialization formats for inter-service communication in distributed systems. *Advances in Cyber-Physical Systems*, *9*(2), 175–180. https://doi.org/10.23939/acps2024.02.175

6. Castillo, D. C., Rosales, J., & Torres Blanco, G. A. (2018). Optimizing binary serialization with an independent data definition format. *International Journal of Computer Applications*, *180*(28), 15–18. https://doi.org/10.5120/ijca2018916670

7. Jackson, S., Cummings, N., & Khan, S. (2024). Streaming technologies and serialization protocols: Empirical performance analysis. *arXiv:2407.13494 [cs.SE]*. https://doi.org/10.48550/arXiv.2407.13494

8. Salhi, H., Odeh, F., Nasser, R., & Taweel, A. (2018). Benchmarking and performance analysis for distributed cache systems: A comparative case study. In *Lecture notes in computer science* (Vol. 10661, pp. 147–163). Springer, Cham. https://doi.org/10.1007/978-3-319-72401-0_11

9. BenchmarkDotNet. (2025). *Overview of BenchmarkDotNet*. Retrieved from https://benchmarkdotnet.org/articles/overview.html

10. Novak, M. *Serialization performance comparison (XML, Binary, JSON, P…)*. Medium. Retrieved from https://medium.com/@maximn/serialization-performance-comparison-xml-binary-json-p-ad737545d227

11. Octo Technology. *Protocol Buffers: Benchmark and mobile*. Retrieved from https://blog.octo.com/protocol-buffers-benchmark-and-mobile

12. Srinivasa, K. G., & Muppalla, A. K. (2015). *Guide to high performance distributed computing: Case studies with Hadoop, Scalding and Spark* (Computer communications and networks, 321 p.). Springer.

13. Holbrook, J., & Haugom, A. (2025). *High-performance distributed applications*. O'Reilly Media, Inc.

14. Tanenbaum, A., & Steen, M. (2016). *Distributed systems: Principles and paradigms* (2nd ed.).

15. Sterling, T., Brodowicz, M., & Anderson, M. (2018). *High performance computing*. Elsevier. https://doi.org/10.1016/C2013-0-09704-6

16. Khlamov, S., et al. (2022). Astronomical knowledge discovery in databases by the CoLiTec software. In *Proceedings of the 12th IEEE ACIT 2022* (pp. 583–586). Ruzomberok, Slovakia. https://doi.org/10.1109/ACIT54803.2022.9913188

17. Faaique, M. (2023). Overview of big data analytics in modern astronomy. *International Journal of Mathematics, Statistics, and Computer Science, 2*, 96–113. https://doi.org/10.59543/ijmscs.v2i.8561

18. Khlamov, S., et al. (2024). Automated data mining of the reference stars from astronomical CCD frames. *CEUR Workshop Proceedings, 3668*, 83–97.

19. Zhernova, P., et al. (2019). Data stream clustering in conditions of an unknown amount of classes. *Advances in Intelligent Systems and Computing, 754*, 410–418. https://doi.org/10.1007/978-3-319-91008-6_41

20. Savanevych, V., et al. (2023). Mathematical methods for an accurate navigation of the robotic telescopes. *Mathematics, 11*(10), 2246. https://doi.org/10.3390/math11102246

21. Troianskyi, V., Kashuba, V., Bazyey, O., et al. (2023). First reported observation of asteroids 2017 AB8, 2017 QX33, and 2017 RV12. *Contributions of the Astronomical Observatory Skalnaté Pleso, 53*, 5–15. https://doi.org/10.31577/caosp.2023.53.2.5

22. Kirichenko, L., et al. (2023). Application of wavelet transform for machine learning classification of time series. In *Lecture notes on data engineering and communications technologies* (Vol. 149, pp. 547–563). https://doi.org/10.1007/978-3-031-16203-9_31

23. Khlamov, S., et al. (2023). Development of the matched filtration of a blurred digital image using its typical form. *Eastern-European Journal of Enterprise Technologies, 1*(9-121), 62–71. https://doi.org/10.15587/1729-4061.2023.273674

24. Khlamov, S., et al. (2022). The astronomical object recognition and its near-zero motion detection in series of images by in situ modeling. In *Proceedings of the 29th IEEE IWSSIP 2022*. https://doi.org/10.1109/IWSSIP55020.2022.9854475

25. Bodyanskiy, Y., Popov, S., Brodetskyi, F., & Chala, O. (2022). Adaptive least-squares support vector machine and its combined learning-self-learning in image recognition task. In *International Scientific and Technical Conference on Computer Sciences and Information Technologies* (pp. 48–51). https://doi.org/10.1109/CSIT56902.2022.10000518

26. Vlasenko, V., et al. (2024). Devising a procedure for the brightness alignment of astronomical frames background by a high frequency filtration to improve accuracy of the brightness estimation of objects. *Eastern-European Journal of Enterprise Technologies, 2*(2-128), 31–38. https://doi.org/10.15587/1729-4061.2024.301327

27. Khlamov, S., et al. (2024). Machine vision for astronomical images using the modern image processing algorithms implemented in the CoLiTec software. In *Measurements and instrumentation for machine vision* (pp. 269–310). https://doi.org/10.1201/9781003343783-12

28. Romanenkov, Y., Mukhin, V., Kosenko, V., et al. (2024). Criterion for ranking interval alternatives in a decision-making task. *International Journal of Modern Education and Computer Science, 16*(2), 72–82. https://doi.org/10.5815/ijmecs.2024.02.06

29. Troianskyi, V., Godunova, V., Serebryanskiy, A., Aimanova, G., & Franco, L., et al. (2024). Optical observations of the potentially hazardous asteroid (4660) Nereus at opposition 2021. *Icarus, 420*, 116146. https://doi.org/10.1016/j.icarus.2024.116146

30. Khlamov, S., Tabakova, I., Trunova, T., & Deineko, Z. (2022). Machine vision for astronomical images using the Canny edge detector. *CEUR Workshop Proceedings, 3384*, 1–10.

ЕФЕКТИВНІСТЬ ВИКОНАННЯ ПОДІЄВО-ОРІЄНТОВАНОЇ РОЗПОДІЛЕНОЇ СИСТЕМИ З ВИКОРИСТАННЯМ ДВІЙКОВИХ КОМУНІКАЦІЙНИХ ПРОТОКОЛІВ

Ph.D. С. Хламов[1][0000-0001-9434-1081], ***С. Орлов***[2][0009-0008-0680-206X], ***Т. Трунова***[3][0000-0003-2689-2679], ***Ph.D. А. Фролов***[4][0000-0001-7335-0712], ***Д. Жужнєв***[5][0009-0001-8778-3241]

Харківський національний університет радіоелектроніки, Україна
EMAIL: [1]sergii.khlamov@gmail.com,[2]stasorlov21@gmail.com, [3]tetiana.trunova@nure.ua,[4]andrii.frolov@nure.ua, [5]zhuzhniev@gmail.com

Анотація. *Розділ присвячено підвищенню ефективності продуктивності шляхом інтеграції бінарних комунікаційних протоколів у подієво-орієнтовані розподілені системи. У ньому стверджується, що хоча традиційні текстові протоколи, такі як JSON, забезпечують зручність читання людиною, вони створюють значні накладні витрати за обсягом даних, швидкістю серіалізації/десеріалізації та використанням мережевої пропускної*

здатності, що перешкоджає ефективній роботі у високонавантажених середовищах. У розділі наведено порівняльний аналіз бінарних протоколів серіалізації, зокрема Protocol Buffers (Protobuf), MessagePack і Apache Avro, у порівнянні з текстовими форматами, такими як JSON. Аналіз базується як на кількісних показниках - швидкості серіалізації, розмірі стисненого повідомлення, так і на якісних - підтримці схем, зворотній сумісності та можливостях потокової чи пакетної обробки даних. Розроблений конвеєр демонструє, що бінарні протоколи забезпечують суттєві переваги у продуктивності, включаючи зниження затримки, збільшення пропускної здатності та значну економію мережевих ресурсів. Наприклад, MessagePack і Protobuf показують набагато вищу швидкість серіалізації/десеріалізації порівняно з JSON, а також формують значно менші за розміром повідомлення. У розділі зроблено висновок, що для високопродуктивних, низьколатентних і високопропускних подієво-орієнтованих систем бінарні протоколи є часто не просто оптимізацією, а фундаментальною необхідністю. Крім того, запропоновано рамкову модель прийняття рішень, яка допомагає розробникам і архітекторам обирати відповідний протокол залежно від конкретних вимог системи, з особливим акцентом на забезпеченні цілісності інтелектуальної власності та захисті від несанкціонованого копіювання.

***Ключові слова**: серіалізація даних, текстові та бінарні формати, JSON, автоматизований конвеєр, продуктивність, швидкість серіалізації даних, розмір серіалізованого повідомлення, мережна затримка, підтримка схем даних і зворотна сумісність, потокова передача подій, подієво-орієнтована архітектура, .NET Core, MessagePack, Protobuf, Apache Avro, BenchmarkDotNet.*

УДК 528.8:004.9:631.4
DOI https://doi.org/10.36059/978-966-397-538-2-11

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ ВИЯВЛЕННЯ ПОКИНУТИХ СІЛЬСЬКОГОСПОДАРСЬКИХ УГІДЬ НА ОСНОВІ СУПУТНИКОВОГО МОНІТОРИНГУ

Ph.D. К. Сергєєва[1][0000-0001-7345-2209], **Ph.D. Ю. Кавац**[2][0000-0002-0180-5957], **Dr.Sci.O. Ковров**[1][0000-0003-3364-119X], **Д. Чумичов**[1][0009-0005-2729-0735]

[1]Національний технічний університет "Дніпровська політехніка", Україна
[2]Український державний університет наук і технологій, Україна
EMAIL: [1]sergieieva.k.l@nmu.one, [2]yukavats@gmail.com, [1]kovrov.o.s@nmu.one,
[1]chumychov.d.d@nmu.one

***Анотація.** Розроблено інформаційну технологію автоматизованого виявлення покинутих сільськогосподарських угідь за часовими рядами даних*